

A preliminary univalent formalization of the p -adic numbers

Álvaro Pelayo

Vladimir Voevodsky

Michael A. Warren

Abstract

The goal of this paper is to give a preliminary formalization of the p -adic numbers, in the context of the Univalent Foundations. We also provide the corresponding code verifying the construction in the proof assistant Coq. Because work in the univalent setting is ongoing, the structure and organization of the construction of the p -adic numbers we give in this paper is expected to change as Coq libraries are more suitably rearranged, and optimized, by the authors and other researchers in the future. So our construction here should be deemed as a first approximation which is subject to improvements.

1 Introduction

In this paper we present a preliminary formalization of the construction of the p -adic numbers in the Coq proof assistant. The formalization is carried out in the *univalent setting* introduced by the second author [19]. This setting, which is based on insights from homotopy theory and higher-dimensional category theory, serves as an overall organizational and methodological framework which informs our construction. At the same time, our construction has several ingredients which are familiar in constructive mathematics. Because work on formalization in this direction is ongoing, the Coq code associated with this paper may be updated accordingly in the future by the authors and others. As such, the structure and content of the Coq code described here may not match exactly the code which is ultimately included in the Univalent Foundations libraries. Readers interested in making use of the code should accordingly consult the latest version available.

We chose to formalize the p -adic numbers as a first step in the development and formalization of the p -adic theory of integrable systems. We hope that this will prove to be a promising approach to this theory which should facilitate progress in the field in the future, in particular with regard to the construction of algorithms and their numerical analysis. Ultimately, we hope that insights from this project could be useful in the setting of real integrable systems.

The idea of the univalent perspective is, roughly, to develop mathematics within the world of homotopy types. By virtue of taking this approach we are able to make use of type theory as a calculus for formal reasoning about homotopy types. We hope that in the

future, because this development of mathematics can be carried out in a proof assistant such as Coq so that the proofs carry some algorithmic content, it will be possible to extract good algorithms from the proofs. One of our motivations is that the construction of such algorithms would in turn help with some problems concerning integrable systems which are of particular interest in applications. For instance, one outstanding problem is: given numerical spectral data about a quantum system (coming from an experiment), extract an algorithm to reconstruct the classical integrable system, see Section 7.

We will only briefly touch upon the technical details of homotopy type theory and the univalence axiom, and we refer the reader to [2] for a basic introduction to homotopy type theory. For univalent foundations and the second author’s Coq library [18] we refer readers to [15], where a description of the research program, its motivations, and its implementation in Coq, are given. Because it is assumed that the reader is already familiar with Coq and with the second author’s program, this paper has been written in a style which we foresee future papers in formalization taking: it is a summary of the Coq code written in ordinary mathematical English. The details are of course in the Coq code, but the overall structure of the formalization (as well as the key steps of the proofs) should be apparent from the sketch given here. The actual Coq code associated to this paper can be found on the websites of the authors, as supplementary files to the arXiv posting of this paper, and as an appendix to the present paper.

Structure of paper

Hensel [10] invented the p -adic numbers \mathbb{Q}_p about one hundred years ago. The p -adic numbers and the reals are the canonical metric completions of the rationals. Classically, there are a number of ways to construct the p -adic numbers, and we refer the reader to [8, 11, 16] for further details regarding the classical theory. The construction of the p -adic numbers given in this paper is constructive and uses algebraic, rather than analytic, techniques. Namely, we first construct the integral domain of p -adic integers \mathbb{Z}_p as a quotient of the ring $\mathbb{Z}[[X]]$ of formal power series over \mathbb{Z} . We were unable to find the specific construction of \mathbb{Z}_p we employ in the literature, but we believe that it is known. We then take the p -adic numbers \mathbb{Q}_p to be the field of fractions of \mathbb{Z}_p . Because we are working constructively, and because $\mathbb{Z}[[X]]$ does not have decidable equality, it is necessary to work with an apartness relation and with the corresponding notions of integral domains and fields. We will refer to the apartness versions of fields as *Heyting fields* following the standard usage in constructive mathematics.

In detail, this paper is organized as follows. In Section 2, we give a brief overview of the univalent setting. In Section 3 we review some basic constructive algebra. Section 4 contains our construction of formal power series and the proofs of several results on formal power series. The proof that it is possible to form the Heyting field of fractions for an integral domain is given in Section 5. The construction of the p -adic numbers appears in Section 6. Section 7 is a brief epilogue containing a sketch of some future plans concerning p -adic integrable systems. Finally, the Coq code can be found in the Appendix 7.2. Although this appendix is quite long, it is the most important part of the paper and so we feel that it is justified to include it here.

We should note that the p -adic numbers are also relevant in the physics literature, see [5] and the references therein. In fact, one of our main motivations in wanting to develop a p -adic theory of integrable systems is to study inverse spectral problems concerning p -adic analogues of real quantum integrable systems. We refer to Section 7.2 for a list of short term plans concerning the p -adic numbers.

2 Univalent basics

The second author's Coq library spans a large portion of mathematics and we make free use of this library. However, for the sake of clarity we will here mention those specific parts of the library which we use in the construction of the p -adic numbers. A survey of the development of univalent mathematics in Coq can be found in [15].

Notation and conventions

In this paper, and in the Coq files, all rings are assumed to be commutative and with 1.

\mathbb{N} denotes the type of natural numbers which is defined as an inductive type in the standard way. In the Coq code \mathbb{N} is denoted by `nat`. Similarly, \mathbb{Z} denotes the type of integers which is constructed as the group completion of the abelian monoid of natural numbers. In the Coq code \mathbb{Z} is denoted by `hz`.

\mathcal{U} denotes a fixed universe of types. In the Coq code this is denoted by `UU`. The identity type $\text{Id}_A(a, b)$ is denoted by $a \rightsquigarrow b$. In the Coq files this is denoted by either `paths a b` or by $a \sim > b$.

We write $\prod_{x:A} .B(x)$ for dependent products and $\sum_{x:A} .B(x)$ for dependent sums (defined here as the record type `total2`).

We will generally use the same naming conventions as used in the Coq files, but in some cases we will introduce abbreviations, such as $\sum_{i=0}^n f(i)$ for summation, when it will improve the readability.

Because the current implementation of the underlying type system of Coq does not handle universes (and several related matters) in a way which is completely suited for the univalent development of mathematics, it is necessary to apply several patches to the Coq system in order to compile the second author's Coq library as well as the files described in this paper. Instructions on how to compile a patched version of Coq can be found in the second author's library.

2.1 Basic homotopy theoretic notions in Coq

We think of \mathcal{U} as the universe of small homotopy types (or fibrant and cofibrant spaces). For $B : \mathcal{U}$, we represent a dependent type over B as a term $E : B \rightarrow \mathcal{U}$. From the perspective of homotopy theory this corresponds to a fibration over B and, for $b : B$, $E(b)$ corresponds to the fiber over b . The dependent product $\prod_{x:B} E(x)$ is regarded as the space of sections of the fibration represented by E . Similarly, the dependent sum, $\sum_{x:B} E(x)$ corresponds to

the total space of the fibration. We think of the identity type $a \rightsquigarrow b$ as denoting the fiber of the path space over (a, b) . We will use the phrases "path space" and "type of paths" interchangeably for this type. I.e., a term $f : a \rightsquigarrow b$ corresponds to a path from a to b .

Given a path $f : b \rightsquigarrow b'$ in B and a point $e : E(b)$ in the fiber over b we obtain a corresponding point $f_!(e) : E(b')$ in the fiber over b' . In the Coq code $f_!$ is denoted by `transportf E f e`. In order to construct a path $x \rightsquigarrow y$ in the total space $\sum_{x:B} E(x)$ it suffices to construct a path $f : \pi_1(x) \rightsquigarrow \pi_1(y)$ and a path $g : f_!(\pi_2(x)) \rightsquigarrow \pi_2(y)$.

Given a term $g : B \rightarrow A$ and a path $f : b \rightsquigarrow b'$ in B , we obtain a path $g(f) : g(b) \rightsquigarrow g(b')$. In the Coq code $g(f)$ is denoted by `maponpaths g f`. This corresponds, regarding a homotopy type as an ∞ -groupoid, the weakly functorial action of g on the path f .

Definition 2.1 (hfiber). Given types A and B , $g : B \rightarrow A$ and $a : A$, the **homotopy fiber of g over a** is the type

$$\mathbf{hfiber}\ g\ a := \sum_{x:B} (g(x) \rightsquigarrow a).$$

Definition 2.2 (iscontr). We define the type **iscontr**(A) of proofs that A is contractible as

$$\mathbf{iscontr}(A) := \sum_{c:A} \prod_{x:A} (x \rightsquigarrow c).$$

We say that A is **contractible** if **iscontr**(A) is inhabited.

We will see below that contractibility in this setting plays the same role as canonical existence in the classical development of mathematics.

Definition 2.3 (isweq and weq). Given $g : B \rightarrow A$ we define the type **isweq**(g) of proofs that g is a weak equivalence as

$$\mathbf{isweq}(g) := \prod_{x:A} \mathbf{iscontr}(\mathbf{hfiber}\ g\ x).$$

If **isweq**(g) is inhabited, then we say that g is a **weak equivalence**.

There is a filtration of types into different "h-levels". Homotopy theoretically this is a slight extension of the usual filtration by homotopy n -types. We will only require the first few h-levels in this paper.

Definition 2.4 (isofhlevel, isaprop, hprop, isaset and hset). A type A is of **h-level**:¹

- 0 if A is contractible;
- $(n + 1)$ if, for all $a, b : A$, the type $(a \rightsquigarrow b)$ is of h-level n .

¹Note that in order to define **isofhlevel** as a type which has values in \mathcal{U} , as is done in the file `uu0.v` from the second author's Coq library, it is necessary to compile Coq with a patch.

We denote by $\iota_n(A)$ the type of proofs that A is of h-level n . We abbreviate $\iota_1(A)$ by **isaProp**(A) and $\iota_2(A)$ by **isaSet**(A). We write **hProp** for the type of (small) types of h-level 1 and **hSet** for the type of (small) types of h-level 2.

Intuitively, **hProp** consists of those spaces which are homotopy equivalent to either the empty space 0 or to the one element space 1. Accordingly, **hProp** plays the role played by the Booleans in classical logic or by the subobject classifier in topos logic. Types in **hProp** satisfy proof-irrelevance (**proofirrelevance**) and, indeed (**invproofirrelevance**), being an h-prop is equivalent to being proof-irrelevant.

Intuitively, **hSet** consists of those spaces which are homotopy equivalent to discrete spaces. I.e., these are the sets. Most of the types which we will be dealing with are either h-props or h-sets. We will sometimes refer to h-sets simply as "sets" when no confusion will result.

We make use of a number of basic properties of h-levels. E.g.,

1. **impred**: for $n : \mathbb{N}$, $B : \mathcal{U}$ and $E : B \rightarrow \mathcal{U}$, the type

$$\prod_{x:B} \mathbf{isofhlevel}_n(E_x) \rightarrow \mathbf{isofhlevel}_n(\prod_{x:B} E_x)$$

is inhabited.

2. **impredfun**: for $n : \mathbb{N}$, $A, B : \mathcal{U}$, if A is of h-level n , then so is $(B \rightarrow A)$.
3. **isofhleveldirprod**: If A is of h-level n and B is of h-level n , then so is $A \times B$.

2.2 Function extensionality

We make extensive use of the principle of function extensionality (**funextfun**), which follows from the second author's *Univalence Axiom*.

Definition 2.5 (**funextfun**). The principle of **function extensionality** states that, for any two functions $f, g : A \rightarrow B$, the type

$$(\prod_{x:A} f(x) \rightsquigarrow g(x)) \rightarrow (f \rightsquigarrow g)$$

is inhabited.

2.3 Properties of hProp

Given a type $A : \mathcal{U}$, there is a universal way to turn A into a h-prop. This is the "inhabited" construction:

Definition 2.6 (`ishinh_UU`). We say that $A : \mathcal{U}$ is **h-inhabited** if the type

$$\hat{A} := \prod_{P:\mathbf{hProp}} ((A \rightarrow P) \rightarrow P)$$

is inhabited.

It is immediate, using the facts about h-levels sketched above to see that \hat{A} is an h-prop. Moreover, there is a projection $\pi_A : A \rightarrow \hat{A}$ given by

$$\pi_A := \lambda_{x:A} \lambda_{P:\mathbf{hProp}} \lambda_{f:A \rightarrow P} f(x).$$

The map π_A is the universal map from A into a h-prop. To see this, observe that if Q is any h-prop and $f : A \rightarrow Q$, then we have a commutative (up to definitional equality) diagram

$$\begin{array}{ccc} \hat{A} & \xrightarrow{\bar{f}} & Q \\ \pi_A \swarrow & & \nearrow f \\ A & & \end{array}$$

where

$$\bar{f} := \lambda_{t:\hat{A}} t(Q)(f).$$

Moreover, since Q is a h-prop it follows (using function extensionality) that the space of such extensions \bar{f} is contractible.

Using the h-inhabited construction it is possible to endow **hProp** with the structure of a Heyting algebra. This structure is summarized below:

Definition 2.7 (`htrue`,`hfalse`,`hconj`,`hdisj`,`hneg`,`himpl`). For $P, Q : \mathbf{hProp}$ and $X, Y : \mathcal{U}$ we define logical operations on **hProp** as follows:

- 1 and 0 are h-props.
- $P \wedge Q := P \times Q$.
- $X \vee Y := \widehat{X + Y}$.
- $\neg X := X \rightarrow 0$.
- $X \implies P := X \rightarrow P$.

In addition to the Heyting algebra operations, there is an existential quantifier (`hexists`) which is defined by

$$\exists_{x:X} P(x) := \widehat{\sum_{x:X} P(x)}$$

for any $P : X \rightarrow \mathcal{U}$ and $X : \mathcal{U}$. This quantifier satisfies the usual properties of the existential quantifier in intuitionistic logic. Note that our \exists does *not* correspond to the built-in existential quantifier "exists" in Coq.

The proof that, with the operations above, **hProp** is a Heyting algebra makes use of the *Propositional Univalence Axiom* (**uahp**) which says that every logical equivalence between h-props induces a path between them. I.e., it says that the type

$$\prod_{P,Q:\text{hProp}} (P \rightarrow Q) \rightarrow ((Q \rightarrow P) \rightarrow (P \rightsquigarrow Q)).$$

is inhabited.

2.4 Set quotients of types

The second author has given several constructions of quotients of types. A **hsubtype** of a type A is given by a map $S : A \rightarrow \text{hProp}$. Denote by $\mathcal{P}(A)$ the type of hsubtypes of A . Given a relation R on A (that is, $R : A \rightarrow A \rightarrow \text{hProp}$), an **equivalence class** consists of a subtype S of A together with the following data:

1. a term of type $\widehat{\sum_{x:A} S(x)}$.
2. a term of type $\prod_{x,y:A} (xRy \rightarrow S(x) \rightarrow S(y))$.
3. a term of type $\prod_{x,y:A} (S(x) \rightarrow S(y) \rightarrow xRy)$.

Given a subtype S , we denote by **iseqclass** $_R(S)$ the type consisting of such data. The **set quotient** A/R (**setquot**) of a type A by a relation R is then defined by

$$A/R := \sum_{S:\mathcal{P}(A)} \text{iseqclass}_R(S).$$

It is shown (**isasetsetquot**) in the second author's library that A/R is a set and that, when R is an equivalence relation, this set has the usual universal property. In particular, there is a function $\pi : A \rightarrow A/R$ (**setquotpr**) which is compatible with the equivalence relation and, for any set B and function $f : A \rightarrow B$ which is compatible with R , there exists an extension \bar{f} making the diagram

$$\begin{array}{ccc} A/R & \xrightarrow{\bar{f}} & B \\ \pi \swarrow & & \searrow f \\ A & & \end{array}$$

commute. We will make free use throughout of the results on set quotients from the second author's library.

3 Basics on constructive algebra

We will here briefly recall some basics of constructive algebra. For a more detailed treatment we refer to [6] and [12].

The usual definitions of fields and integral domains are not entirely satisfactory from the perspective of constructive algebra since they deal with negative properties (the property of being a non-zero element of the field). From the constructive perspective, it is more appropriate to replace the notion of an element x being non-zero ($x \neq 0$) with x being **apart from zero**, written $x \# 0$.

We will now recall the basics regarding apartness relations.

Definition 3.1. (isapart) A relation $R : \mathbf{hRel}(X)$ is an **apartness relation** provided that it satisfies the following conditions:

Irreflexive for all $x : X$, $\neg(xRx)$.

Symmetric for all $x, y : X$, xRy implies yRx .

Cotransitive for all $x, y : X$, if xRy , then either xRz or zRy , for any $z : X$.

Classically, the negation of equality $x \neq y$ relation is an apartness relation. However, negation of equality is not the only classical apartness relation. For example, if X is a topological space, then the relation R given by xRy if and only if x and y are in different connected components is an apartness relation. (This example can be generalized to give a limitless number of classical examples of apartness relations.)

For $X : \mathbf{hSet}$, we denote by $\mathbf{Apart}(X)$ the type of apartness relations on X . We generally denote apartness relations by $x \# y$. When a type has decidable equality the negation of equality is an apartness relation:

Lemma 3.2 (deceqtoeqapart). *If $X : \mathbf{hSet}$ has decidable equality, then negation of equality*

$$\neg(x \rightsquigarrow y)$$

is an apartness relation on X .

Definition 3.3 (isapartdec). Given $X : \mathbf{hSet}$ and $R : \mathbf{Apart}(X)$, we say that R is a **decidable apartness relation on X** if the type

$$(aRb) + (a \rightsquigarrow b)$$

is inhabited.

It is immediate (isapartdectodeceq) that if R is a decidable apartness relation on X , then X has decidable equality.

When we are considering algebraic structures equipped with apartness relations we will require that the relation is compatible with the operations under consideration. In particular, for rings we have the following.

Definition 3.4 (`acommrng`). The type `aCRng` consists of commutative rings A together with an apartness relation $x \# y$ on A which is compatible with the ring structure of A in the sense that²

- (i) For all $a, b, c : A$, if $(c + a) \# (c + b)$, then $a \# b$.
- (ii) For all $a, b, c : A$, if $(c \cdot a) \# (c \cdot b)$, then $a \# b$.

When a commutative ring A has decidable equality it is straightforward to verify that negation of equality is compatible with the ring operations in the sense of Definition 3.4.

Definition 3.5 (`aintdom`). The type `aDom` consists of $A : \text{aCRng}$ such that

- $1 \# 0$.
- For all $a, b : A$, if $a \# 0$ and $b \# 0$, then $(a \cdot b) \# 0$.

We refer to the terms of type `aDom` as **apartness domains**.

Heyting fields are the appropriate generalization of fields to the constructive setting when one considers algebraic structures with apartness relations:

Definition 3.6 (`afld`). The type `aFId` of **Heyting fields** consists of $A : \text{aCRng}$ such that

- $1 \# 0$.
- For all $a : A$, if $a \# 0$, then a has a multiplicative inverse (the type of multiplicative inverses of a is inhabited).

We have the following immediate observation:

Lemma 3.7 (`afldtoaintdom`). *If A is a Heyting field, then A is an apartness domain.*

Proof. It is immediate to prove that, in a Heyting field, if a has a multiplicative inverse, then it is apart from 0 (`afldinvertibletozero`). It follows that $1 \# 0$. One can show that if a and b both possess multiplicative inverses, then so does their product $a \cdot b$ (`multinvmultstable`). It is then immediate that $(a \cdot b) \# 0$ when $a \# 0$ and $b \# 0$. \square

4 Formal power series

Our treatment of formal power series makes use of function extensionality, since formal power series over a commutative ring R are here defined as terms of type $\mathbb{N} \rightarrow R$ with the operations of addition and multiplication given in the usual way. The main result of this section is that, with these operations, formal power series is a commutative ring. Moreover, there is a natural apartness relation on formal power series and, furthermore, when the ring R has decidable equality the ring of formal power series over R forms an apartness domain. We will now fill in the details of this sketch.

²Note that in the Coq files we actually require the corresponding cancellation properties also on the right. This is redundant for commutative rings, but for general rings one requires also these further properties.

4.1 Summation in a ring

We define both a restrictive summation operation (`natsummation0`), which allows us to form the sum $\sum_{i=0}^n a_i$ of a sequence $a : \mathbb{N} \rightarrow R$, and a more general operation (`summation`), which allows us to form the sum $\sum_{i=m}^n a_i$ of a sequence $a : \mathbb{Z} \rightarrow R$. However, we will only really require the former of these two constructions and so we will omit details related to the more general summation. In order to avoid confusion with our notation for dependent sums, we write $\bigoplus_{i=0}^n a_i$ for the sum $\sum_{i=0}^n a_i$. Summation is, of course, defined inductively by setting

$$\bigoplus_{i=0}^0 a_i := a_0 \quad \text{and} \quad \bigoplus_{i=0}^{n+1} a_i := \left(\bigoplus_{i=0}^n a_i \right) + a_{n+1}.$$

Manipulation of sums

It is important to note that when we manipulate sums, to obtain new sums, *what is relevant is that there is a path between them, and not whether they are equal in the strict sense*. This is a crucial point which underlies in a fundamental way much of the univalent approach to mathematics. The following lemma includes several basic facts regarding the behavior of summation of which we will make frequent use:

Lemma 4.1. *Given a natural number n and sequences $a, b : \mathbb{N} \rightarrow R$, we have the following:*

1. (`natsummationpathsupperfixed`) *Given $p : \prod_{x:\mathbb{N}} (x \leq n) \rightarrow (a_x \rightsquigarrow b_x)$, the type*

$$\bigoplus_{i=0}^n a_i \rightsquigarrow \bigoplus_{i=0}^n b_i$$

is inhabited.

2. (`natsummationshift0`) *The type*

$$\bigoplus_{i=0}^{n+1} a_i \rightsquigarrow \left(\bigoplus_{i=0}^n a_{i+1} \right) + a_0$$

is inhabited.

In order to more easily handle reindexing of sums we introduce, for $f : \mathbb{N} \rightarrow \mathbb{N}$, the type $\mathbf{Aut}_n(f)$ (`isnattruncauto`) of proofs that f is an automorphism of the interval $[0, n]$ of natural numbers. Explicitly, $\mathbf{Aut}_n(f)$ is defined to be the following type:³

$$\left(\prod_{x \leq n} \sum_{y \leq n} ((f(y) \rightsquigarrow x) \times \prod_{z \leq n} (f(z) \rightsquigarrow x) \rightarrow (y \rightsquigarrow z)) \right) \times \left(\prod_{x \leq n} (f(x) \leq n) \right)$$

³Note that we could, alternatively, have used the type $(\prod_{x \leq n} \sum_{y \leq n} (f(y) \rightsquigarrow x)) \times (\prod_{x \leq n} (f(x) \leq n))$. However, the more verbose type we give here is convenient, for purposes of formalization, as it allows for more direct proofs of subsequent lemmas.

where we have abbreviated $\prod_{x:\mathbb{N}} (x \leq n) \rightarrow \dots$ as $\prod_{x \leq n} \dots$ and $\sum_{x:\mathbb{N}} (x \leq n) \times \dots$ as $\sum_{x \leq n} \dots$. It is possible to reindex sums along such automorphisms, as shown by the following lemma:

Lemma 4.2. (natsummationreindexing) *Given a natural number n and a map $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{Aut}_n(f)$ is inhabited, the type*

$$\bigoplus_{i=0}^n a_i \rightsquigarrow \bigoplus_{i=0}^n a_{f(i)}$$

for any sequence $a : \mathbb{N} \rightarrow R$, is inhabited.

The final fact regarding summation which we require is the following:

Lemma 4.3. (natsummationswap) *Given $f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow R$ and a natural number n , the type*

$$\bigoplus_{k=0}^n \bigoplus_{l=0}^k f(l, k-l) \rightsquigarrow \bigoplus_{k=0}^n \bigoplus_{l=0}^{n-k} f(k, l)$$

is inhabited.

4.2 The ring of formal power series

We define, for a type A , the type of sequences of elements of A (`seqson`) as the function space $\mathbb{N} \rightarrow A$. When A is a set so is $\mathbb{N} \rightarrow A$ and for A a commutative ring we take $\mathbb{N} \rightarrow A$ as the underlying set (`fps`) of the ring of formal power series over A . If a is a sequence on A , then we write $a_n : A$ for the result of evaluating the sequence at the natural number n .

Ring operations on formal power series

For a given commutative ring R , addition and multiplication of formal power series are defined as usual by the formulae:

$$(a + b)_n := a_n + b_n$$

$$(a \cdot b)_n := \bigoplus_{k=0}^n a_k b_{n-k}.$$

The zero sequence 0 is given by $0_n := 0$ for all natural numbers n and the sequence 1 is given by $1_0 := 1$ and $1_{n+1} := 0$ for all natural numbers n .

Proposition 4.4 (fpscommrng). *Let $(R, +, \cdot)$ be a commutative ring. Then the set of sequences on R with the operations given above is a commutative ring.*

Proof. The proof follows from the facts about summation described above. For example, to prove associativity of multiplication, we must show that, for all natural numbers n ,

$$\bigoplus_{i=0}^n \left(\bigoplus_{k=0}^i a_k \cdot b_{i-k} \right) \cdot c_{n-i} \rightsquigarrow \bigoplus_{j=0}^n a_j \cdot \left(\bigoplus_{l=0}^{n-j} b_l \cdot c_{(n-j)-l} \right).$$

For this, we reason as follows

$$\bigoplus_{j=0}^n \bigoplus_{l=0}^{n-j} a_j \cdot (b_l \cdot c_{(n-j)-l}) \rightsquigarrow \bigoplus_{l=0}^n \bigoplus_{j=0}^l (a_l \cdot b_{k-l}) \cdot c_{n-l-(k-l)} \rightsquigarrow \bigoplus_{l=0}^n \bigoplus_{j=0}^l a_l \cdot (b_{k-l} \cdot c_{n-k}),$$

where the first path is given by Lemma 4.3 and associativity of multiplication in R . In the Coq proof this line of reasoning is put together with generous use of Lemma 4.1, (`funextfun`), several minor lemmas such as (`natsummationtimesdist1`), and associativity of R itself. \square

4.3 The apartness relation on formal power series

Although it is not used in the construction of the p -adic numbers, we mention here some results contained in the Coq files regarding apartness relations on formal power series.

Assume that R is a commutative ring with an apartness relation. Then there is an induced apartness relation on $R[[X]]$ given by setting (`fpsapart`)

$$a \# b \quad \text{if and only if} \quad \exists_{n:\mathbb{N}}. a_n \# b_n \tag{1}$$

for $a, b : R[[X]]$. This apartness relation is compatible with the ring operations and so we see that $R[[X]] : \mathbf{aCRng}$ (`acommrngfps`).

For R an apartness domain, provided that the apartness relation on R is decidable in the sense of Definition 3.3, it is possible to show that $R[[X]]$ is an apartness domain.

Proposition 4.5 (`apartdectoisaintdomfps`). *For $R : \mathbf{aDom}$ with decidable apartness, the commutative ring $R[[X]]$ of formal power series is an apartness domain when equipped with the apartness relation (1).*

The proof of Proposition 4.5 is a consequence of the following lemma:

Lemma 4.6 (`leadingcoefficientapartdec`). *For $R : \mathbf{aDom}$ and $a : R[[X]]$, if $a_0 \# 0$, then for any $n : \mathbb{N}$ and $b : R[[X]]$, if $b_n \# 0$, then $(a \cdot b) \# 0$.*

Proof. The proof is by induction on n and is obvious in the base case. The induction case splits into two subcases depending on whether $b_0 \# 0$ or $b_0 \rightsquigarrow 0$. In the former case, $(a \cdot b)_0 \# 0$, whereas in the latter case the claim follows by applying the induction hypothesis to the sequence $b' : R[[X]]$ given by $b'_n := b_{n+1}$. \square

5 The Heyting field of fractions

The construction of the Heyting field of fractions from an apartness domain is a classical result in constructive algebra due to Heyting and we therefore give only a brief sketch of the details here.

Definition 5.1 (`aintdomzerosubmonoid`). Given $A : \mathbf{aDom}$, we denote by \tilde{A} the sub-monoid of A (with respect to the multiplicative structure of A) consisting of those $a : A$ such that $a \# 0$.

It follows (`commrngfrac`) that there exists a commutative ring $A[\tilde{A}^{-1}]$ obtained by localizing with respect to \tilde{A} . It remains to show there exists an apartness relation on $A[\tilde{A}^{-1}]$ which makes it into a Heyting field.

Definition 5.2 (`afldfracapartrel0`). For elements $a, c : A \times \tilde{A}$ we define

$$a \# c \quad \text{if and only if} \quad ((\pi_1 a) \cdot (\pi_2 c)) \# ((\pi_1 c) \cdot (\pi_2 a)).$$

This relation extends to a relation (`afldfracapartrel`) on $A[\tilde{A}^{-1}]$ and it is straightforward to show that it is an apartness relation (`afldfracapart`) which is compatible with the ring structure of $A[\tilde{A}^{-1}]$ (`afldfrac0`). For instance (`iscotransafldfracapartrelpre`), to see that it is cotransitive suppose given $(a, a') \# (c, c')$ and some (b, b') . Then, by the fact that A is an apartness domain, we see that $a \cdot c' \cdot b' \# c \cdot a' \cdot b'$. Therefore, by cotransitivity of the apartness relation of A , we have that either $a \cdot c' \cdot b' \# b \cdot a' \cdot c'$ or $b \cdot a' \cdot c' \# c \cdot a' \cdot b'$. In the former case it follows that $a \cdot b' \# b \cdot a'$. I.e., $(a, a') \# (b, b')$. In the latter case it similarly follows that $(b, b') \# (c, c')$.

Given $a \in A \times \tilde{A}$ such that $a \# 0$, we have $\pi_1(a) \# 0$ and therefore, we take a^{-1} to be given by the pair $(\pi_2(a), \pi_1(a))$. This definition extends to a definition of the inverse of an element apart from 0 in $A[\tilde{A}^{-1}]$ and it is straightforward to show that this gives makes $A[\tilde{A}^{-1}]$ a Heyting field:

Theorem 5.3 (`afldfracisafld`). *For $A : \mathbf{aDom}$, with the definitions given above, $A[\tilde{A}^{-1}]$ forms a Heyting field.*

We refer to the Heyting field from Theorem 5.3 as the **Heyting field of fractions** of A and we write $\mathbf{Frac}(A)$ for it.

6 The p -adic numbers

The p -adic numbers were invented about one hundred years ago by German mathematician K. Hensel.

6.1 Basic number theory

The following definition is the relation of integer divisibility, and is given as a two part definition in the Coq file. The first part says that, given three integers n, m, k , if the product of n and k is m , then n divides m . The general definition starts only with n and m , and appeals to the existence of k .

Definition 6.1 (`hzdiv0` and `hzdiv`). Let n and m be integers. We write $n|m$ for the type

$$n|m := \exists_{k:\mathbb{Z}}. (m \rightsquigarrow n \cdot k)$$

and we say that n **divides** m when $n|m$ is inhabited.

The division algorithm is then shown to hold via a series of steps. First, we prove the division algorithm for natural numbers. Recall that `pr1` and `pr2` are defined as projections onto the base and “specialization” to a fiber:

Lemma 6.2 (`divalgorithmnonneg`). *For n and m of type `nat`, with m nonzero, there exists a term $qr : (\mathbb{Z} \times \mathbb{Z})$ such that there is a term of type*

$$n \rightsquigarrow (m \cdot \pi_1(qr)) + \pi_2(qr)$$

and there are proofs that $0 \leq \pi_2(qr) < m$.

The proof of Lemma 6.2 is by induction on n with, in the successor step, a case analysis on whether $(r' + 1) < m$ or $r' \rightsquigarrow m$ (that such a case analysis is possible follows from decidability of equality using `hzlehchoice` from the second author’s library). The proof of the general division algorithm is then done by a detailed case analysis (on whether n and m are negative, non-negative or propositionally equal to 0):

Theorem 6.3 (`divalgorithmexists`). *For n and m of type \mathbb{Z} with $m > 0$, the space of terms $qr : \mathbb{Z} \times \mathbb{Z}$ such that the types $n \rightsquigarrow (m \cdot \pi_1(qr)) + \pi_2(qr)$ and $0 \leq \pi_2(qr) < |m|$ are inhabited is contractible.*

Here, as throughout, *contractibility* corresponds to *unique existence* in the traditional setting. One consequence of the division algorithm is that we obtain the operations of taking the quotient and remainder of an integer modulo a non-negative integer (`hzquotientmod` and `hzremaindermod`). These two operations will play a role in a number of calculations in the sequel.

In addition to the division algorithm we also obtain the familiar Euclidean algorithm (again stated in terms of contractibility of an appropriate space):

Theorem 6.4 (`euclideanalgorithm`). *Let n and m be integers with $n \neq 0$. Then the space $\text{hzgcd}(n, m)$ of greatest common divisors of n and m is contractible.*

We also obtain a form of the Bézout lemma:

Lemma 6.5 (bezoutstrong). *For all $m, n : \mathbb{Z}$ such that n is non-zero, the type of $ab : \mathbb{Z} \times \mathbb{Z}$ for which there exists a term of type $\text{gcd}(n, m) \rightsquigarrow \pi_1(ab) \cdot n + \pi_2(ab) \cdot m$ is inhabited.*

Given $p : \mathbb{Z}$, the type of proofs that p is a prime is defined by setting

$$\text{isaprime}(p) := (1 < p) \times ((m|p) \rightarrow (m \rightsquigarrow 1) \vee (m \rightsquigarrow p)).$$

As a consequence of Lemma 6.5 we obtain

Theorem 6.6 (acommrng_hzmod and ahzmod). *For non-zero p of type \mathbb{Z} , $\mathbb{Z}/p\mathbb{Z}$ is a commutative ring with compatible apartness relation. When p is a prime, $\mathbb{Z}/p\mathbb{Z}$ is a Heyting field.*

Note that the apartness relation on $\mathbb{Z}/p\mathbb{Z}$ is the one induced by the fact that equality of $\mathbb{Z}/p\mathbb{Z}$ is decidable (`isdeceqhzmodp`).

6.2 The construction of \mathbb{Q}_p

Throughout this section we assume given a prime p . Explicitly, we require the proof witnessing the fact that p is a prime. We note though that for some of the results stated here it is only necessary that p be non-zero. We also introduce some notation for quotients and remainders modulo p . We denote by $\{a\}$ the quotient of a modulo p (`hzquotientmod`) and by $[a]$ the remainder of a modulo p .

We will now summarize our construction of the apartness domain \mathbb{Z}_p of p -adic integers.

Definition 6.7 (precarry). Given a formal power series a over \mathbb{Z} , we define a new formal power series $\mathbf{p}(a)$ over \mathbb{Z} inductively by

$$\begin{aligned}\mathbf{p}(a)_0 &:= a_0 \\ \mathbf{p}(a)_{n+1} &:= a_{n+1} + \{\mathbf{p}(a)_n\}.\end{aligned}$$

Definition 6.8 (carry). Given a formal power series a over \mathbb{Z} , we define a new formal power series a^\natural over \mathbb{Z} by

$$(a^\natural)_n := [\mathbf{p}(a)_n].$$

We call a^\natural the **carried power series** of a .

Example 6.9. The formal power series $a = (4, 1, 8, 0, \dots)$ is sent to $\mathbf{p}(a) = (4, 2, 8, 2, 0, \dots)$ and to $a^\natural = (1, 2, 2, 2, 0, \dots)$.

The operation of carrying (mod p) for power series induces an equivalence relation \sim (`carryequiv`) on $\mathbb{Z}[[X]]$ by setting

$$a \sim b \quad \text{if and only if} \quad a^\natural \rightsquigarrow b^\natural.$$

Observe that $X - p \sim 0$. Furthermore, for any $a \in \mathbb{Z}[[X]]$, if $a \sim 0$, then there exist integers λ_i such that $a_0 = -\lambda_0 p$ and $a_{n+1} = -\lambda_{n+1} p + \lambda_n$. Using these facts it follows that \sim is the equivalence relation corresponding to the ideal $(X - p)$ in $\mathbb{Z}[[X]]$. Ultimately, once the theory of ideals has been developed in the Univalent Foundations Library, \mathbb{Z}_p will be constructed as the quotient of $\mathbb{Z}[[X]]$ by this ideal. However, because quotients of rings are given in the second author's library in terms of congruences, we here describe \mathbb{Z}_p using the corresponding congruence \sim .

We will now describe the proof that this relation is a congruence with respect to the ring operations on $\mathbb{Z}[[X]]$.

Lemma 6.10 (`quotientprecarryplus`). *For formal power series a and b over \mathbb{Z} ,*

$$\{\mathbf{p}(a + b)_n\} \rightsquigarrow \{\mathbf{p}(a)_n\} + \{\mathbf{p}(b)_n\} + \{\mathbf{p}(a^\natural + b^\natural)_n\}$$

for $n : \mathbb{N}$.

Proof. The proof is by induction on n . In the base case it is trivial and in the induction case it is by the following calculation:

$$\begin{aligned} \{\mathbf{p}(a + b)_{n+1}\} &\rightsquigarrow \{\mathbf{p}(a)_{n+1} + \mathbf{p}(b)_{n+1} + \{\mathbf{p}(a^\natural + b^\natural)_n\}\} \\ &\rightsquigarrow \{\mathbf{p}(a)_{n+1}\} + \{\mathbf{p}(b)_{n+1}\} + \{\mathbf{p}(a^\natural + b^\natural)_n\} + \{a_{n+1}^\natural + b_{n+1}^\natural + [\mathbf{p}(a^\natural + b^\natural)_n]\} \\ &\rightsquigarrow \{\mathbf{p}(a)_{n+1}\} + \{\mathbf{p}(b)_{n+1}\} + \{\mathbf{p}(a^\natural + b^\natural)_{n+1}\} \end{aligned}$$

where the first path is by definition of precarry and the induction hypothesis, the second path is by the familiar decomposition of the quotient of a sum, and the final path is by definition and the fact that the quotient of a remainder is zero. \square

The following observation is a consequence of Lemma 6.10.

Lemma 6.11 (`carryandplus`). *For a and b formal power series over \mathbb{Z} , $(a+b)^\natural \rightsquigarrow (a^\natural + b^\natural)^\natural$.*

Similarly, a straightforward induction gives us the following lemma:

Lemma 6.12 (`precarryandtimes1`). *Given formal power series a and b over \mathbb{Z} ,*

$$\{\mathbf{p}(a \cdot b)_n\} \rightsquigarrow (\{\mathbf{p}(a)\} \cdot b)_n + \{\mathbf{p}(a^\natural \cdot b)_n\}$$

for $n : \mathbb{N}$.

The proof that carrying is compatible with multiplication of power series is then an immediate consequence of Lemma 6.12:

Lemma 6.13 (`carryandtimes`). *Given formal power series a and b over \mathbb{Z} , $(a \cdot b)^\natural \rightsquigarrow (a^\natural \cdot b^\natural)^\natural$.*

It follows from Lemmas 6.11 and 6.13 that the quotient of $\mathbb{Z}[[X]]$ by the equivalence relation \sim is itself a commutative ring (`commrngofpadicints`). Indeed, it is the commutative ring \mathbb{Z}_p of **p -adic integers**. Moreover, there is an apartness relation (`padicapart`) on p -adic integers obtained as the extension of the relation (`padicapart0`)

$$a \# b \quad \text{if and only if} \quad \exists_{n:\mathbb{N}}. \neg(a_n^\natural \rightsquigarrow b_n^\natural), \quad (2)$$

for $a, b : \mathbb{Z}[[X]]$, to the p -adic integers. This apartness relation is straightforwardly seen to be compatible with the ring structure of \mathbb{Z}_p (`acommrngofpadicints`).

Theorem 6.14 (`padicintsareintdom, padicintegers`). *The commutative ring \mathbb{Z}_p with the apartness relation described above forms an apartness domain.*

Proof. It suffices to prove that for $a, b : \mathbb{Z}[[X]]$ such that $a \# 0$ and $b \# 0$ it follows that $a \cdot b \# 0$, where we are considering only the apartness relation (2). Since \mathbb{Z} has decidable equality, it follows (`leastelementprinciple`) that there are natural numbers k and m which are the least natural numbers such that $\neg(a_k^\natural \rightsquigarrow 0)$ and $\neg(b_m^\natural \rightsquigarrow 0)$, respectively. It then follows that $\neg((a \cdot b)_{k+m}^\natural \rightsquigarrow 0)$.

To see this, assume for a contradiction that there is a path $(a \cdot b)_{k+m}^\natural \rightsquigarrow 0$ and consider first the case where $k + m = 0$. Then we have that $a_0 \cdot b_0$ is congruent to 0 modulo p and therefore, since p is prime, either a_0 is congruent to 0 modulo p or b_0 is congruent to 0 modulo p . In either case we have obtained a contradiction.

On the other hand, when $k + m$ is a successor $k + m = n + 1$, we have that

$$(a \cdot b)_{k+m}^\natural \rightsquigarrow [(a^\natural \cdot b^\natural)_{k+m} + \{\mathbf{p}(a^\natural \cdot b^\natural)_n^\natural\}]. \quad (3)$$

By the choice of k and m it follows that there is a further term (`precarryandzeromult`) of type $\mathbf{p}(a^\natural \cdot b^\natural)_n^\natural \rightsquigarrow 0$. Therefore, we obtain a term of type

$$0 \rightsquigarrow [(a^\natural \cdot b^\natural)_{k+m}].$$

However, it is easy (`hzfpstimeswhenzero`) to see that $(a^\natural \cdot b^\natural)_{k+m} \rightsquigarrow a_k^\natural \cdot b_m^\natural$. So, since p is prime, either $a_k^\natural \rightsquigarrow 0$ or $b_m^\natural \rightsquigarrow 0$ is inhabited. In either case we obtain a contradiction. \square

Using Theorem 6.14, we now arrive at our definition of the p -adic numbers:

Definition 6.15 (`padics`). The Heyting field \mathbb{Q}_p of p -adic numbers is defined as the Heyting field of fractions of \mathbb{Z}_p :

$$\mathbb{Q}_p := \mathbf{Frac}(\mathbb{Z}_p).$$

7 Future directions: towards p -adic integrable systems

Next we present an outline of the work on p -adic integrable systems that we plan to carry out following this paper. The long term goal is to develop an analogue of the symplectic theory of finite-dimensional real integrable systems in [13, 14] for p -adic integrable systems in the univalent setting, and implement it in Coq.

We are beginning to explore this, and what we give next is a brief and informal glimpse of our plans. At this point this section is a discussion without rigorous descriptions as we are not yet convinced of the optimal definition of p -adic integrable system. We hope to convey the fact that the p -adic and real theories are expected to be different, and draw attention to the topic; in fact, we are not aware of a uniform treatment of p -adic integrable systems in the symplectic setting.

7.1 Definition of p -adic integrable systems

A word on the contrast between p -adic and real notions

We refer to [8, Section 3] for basic algebraic and topological aspects concerning the p -adic numbers. Many aspects do not match the intuition we have for the real numbers. For instance, there are no nontrivial connected sets and there are non-empty sets which are both compact and open. Other aspects are more familiar: on \mathbb{Q}_p there is an absolute value $|\cdot|$ and \mathbb{Q}_p is complete with respect to it, and there is an inclusion $\mathbb{Q} \rightarrow \mathbb{Q}_p$ with dense image. Continuity and differentiability of functions is defined in the usual way [8, Definitions 4.2.1, 4.2.2]. Continuous functions are uniformly continuous on compact sets, as in the real case.

The notions of continuity and differentiability extend to functions $f: U \subset (\mathbb{Q}_p)^n \rightarrow \mathbb{Q}_p$ of several variables (x_1, \dots, x_n) on open sets U of the Cartesian product $(\mathbb{Q}_p)^n$, in direct analogy with the real case, and in particular we have analogous definitions for partial derivatives $\frac{\partial f}{\partial x_i}$, for all $i = 1, \dots, n$. But although the definitions are the same, differentiability behaves differently in the p -adic case than in the real case. For instance, there are functions $f: \mathbb{Q}_p \rightarrow \mathbb{Q}_p$ which have zero derivative everywhere but are *not* locally constant. Also, the natural extension of the real mean value theorem to the p -adic case is false in general (although a version holds for sufficiently close points), as seen for instance by considering $f(x) = x^p - x$ between the extreme points $a = 0$ and $b = 1$. In this case, [8, Proposition 4.2.3] $f'(x) = px^{p-1} - 1$ and $f(a) = f(b) = 0$ and it is easy to check that any element “in between” a and b , that is, of the form $at + b(1-t) = 1 - t$ for some t with $|t| \leq 1$, gives rise to a unit $f'(1-t)$ in \mathbb{Z}_p .

These differences are an indication that the theory of p -adic integrable systems is not expected to be a direct extension of the theory of real integrable systems, even if the basic definitions are analogous. One can explore such theory classically only, but we hope to do it in the univalent setting, building on the constructions of \mathbb{Q}_p which we have given in the previous sections.

Integrable systems

We are here going to propose a notion of p -adic integrable systems in parallel with the commonly accepted notion of real integrable systems, at least in symplectic geometry.

Because in the univalent foundations, and in Coq, it is nontrivial to define manifolds, for now we are going to work with the p -adic Cartesian product

$$M := (\mathbb{Q}_p)^{2n} = \mathbb{Q}_p \times \dots \text{(2n times)} \dots \times \mathbb{Q}_p$$

with coordinates $(x_1, y_1, \dots, x_n, y_n)$. In this way, we also avoid a discussion of differential or symplectic forms. Fix a p -adic measure on \mathbb{Q}_p , and endow M with the induced product measure.

On M we may consider differentiable functions in the p -adic sense⁴. The following is the formal extension of the definition of real integrable system in finite dimensions. There is, however, a critical point which is not clear to us at the moment, and that's why we restrict our definition to analytic maps, see Remark 7.2.

Definition 7.1. We will say that a (p -adic) analytic map $F := (f_1, \dots, f_n): M \rightarrow (\mathbb{Q}_p)^n$ is a **p -adic integrable system** if two conditions hold:

1. the collection f_1, \dots, f_n satisfies Hamilton's equations:

$$\sum_{k=1}^n \frac{\partial f_i}{\partial x_k} \frac{\partial f_j}{\partial y_k} - \frac{\partial f_i}{\partial y_k} \frac{\partial f_j}{\partial x_k} = 0, \quad \forall 1 \leq i \leq j \leq n. \quad (4)$$

2. the set where the n formal differentials

$$dp_i := \left(\frac{\partial f_i}{\partial x_1}, \dots, \frac{\partial f_i}{\partial x_n}, \frac{\partial f_i}{\partial y_1}, \dots, \frac{\partial f_i}{\partial y_n} \right), \quad \forall 1 \leq i \leq n$$

are linearly dependent has p -adic measure 0.

That is, there exists a p -adic measure 0 set A such that df_1, \dots, df_n are linearly independent on $M \setminus A$. The points where df_1, \dots, df_n are linearly dependent are called *singularities*.

Remark 7.2. This remark explains why we have to restrict to analytic functions in Definition 7.1, when in the real theory one likes to include all smooth functions in the definition of integrable system. There are many interesting, nontrivial p -adic functions that are smooth and have zero derivative everywhere. *However this is not possible if one restricts to analytic functions.* Therefore if f is a smooth solution to a linear differential equation, we could add to f any of these nontrivial functions with zero derivative and obtain a new solution. It follows that all collections of n smooth functions f_1, \dots, f_n which are smooth and have zero derivative everywhere would also form a kind of integrable system, but a very "degenerate" one (in the sense that the differentials df_1, \dots, df_n would not be linearly independent almost

⁴for now we are thinking only of polynomials on $2n$ -variables, which are easy to deal with in Coq.

everywhere as it is normally required for real integrable systems). So this undesirable case does not occur. However, adding functions with zero derivative to an existing system would be unavoidable, giving rise to a new, seemingly very different, p -adic integrable system. We currently understand neither what this means geometrically, nor what it implies for the development of the theory.

7.2 Future plans

The following is a rough outline of what we would like to do next.

Towards p -adic symplectic geometry

- ▷ *p -adic manifolds:* formalize the notion of p -adic manifold in the univalent Foundations with Coq. Formalize Serre's theorem [17] classifying compact p -adic manifolds.
- ▷ *p -adic symplectic forms:* a p -adic symplectic form ω may be defined as in the real case. The closedness condition $d\omega = 0$ makes sense in the p -adic setting, and so does the non-degeneracy condition (in fact, over any field). In the real setting, a theorem of Darboux says that all symplectic forms are locally equivalent, so real symplectic manifolds have no local invariants. It is natural to wonder whether this result holds in the p -adic setting "as is". Because of our previous discussion (see Remark 7.2) one should probably restrict to the analytic setting since $d\omega = 0$ is in fact a system of partial differential equations. Darboux's theorem plays a leading role in the theory of real integrable systems.

Towards p -adic integrable systems: basic theory

- ▷ *construction of p -adic integrable systems:* define p -adic integrable systems on p -adic manifolds, not just $(\mathbb{Q}_p)^n$, and implement this in the univalent foundations using Coq.
- ▷ *p -adic local and semiglobal theory:* develop the local and semilocal theory of p -adic integrable systems in Coq. The local theory basically refers to local models, and the semilocal theory refers to local models in neighborhoods of fibers. One is interested in both the topological and symplectic classification of such models. We are not aware of results describing the topological, or symplectic, structure of regular or singular fibers in the p -adic setting.

In the real case, the regular fibers and their neighborhoods are understood (this is the famous Action-Angle Theorem due to Mineur and Arnold.) The singular fibers may be complicated and not yet well understood in the real setting either (if one restrict to the real analytic setting, then the theory is better understood).

Towards p -adic toric and semitoric systems

- ▷ *p -adic toric systems:* a particular class of real integrable systems which has been thoroughly studied and is well understood, is that of toric integrable systems $F = (f_1, \dots, f_n)$ on $2n$ -dimensional compact symplectic manifolds (M, ω) . These are systems in which each component f_i generates a flow which is periodic of a fixed period. In this case, F is called a *momentum map*. Atiyah [1], Guillemin-Sternberg [9] and Delzant [7] proved a series of striking theorems concerning these systems in the 1980s, which in particular led to complete combinatorial classification in terms of convex polytopes by Delzant (these convex polytopes are nothing by the images of M under F). A theorem of Serre [17] classifies compact p -adic analytic varieties. If on these varieties we would consider actions of the p -adic n -torus, we do not know to what extent the above results could be extended. If in Definition 7.1 one allows smooth non-analytic functions, these results would not hold (see Remark 7.2).
- ▷ *p -adic semitoric systems:* give a classification of p -adic integrable systems under some periodicity condition in analogy with [13, 14].

Spectral questions for p -adic integrable systems

Here we restrict to the systems in the previous section, for which we know that in the real case a full classification may be given.

- ▷ *Inverse spectral problems:* construct algorithms to solve inverse spectral problems about quantum integrable systems. The leading question in the real case is: given the spectrum, can one recover the system from it?
- ▷ *Numerical implementation of inverse spectral problems:* constructing numerically accurate algorithms to solve inverse spectral problems.

The first subsection above should be within reach. We expect the second and third subsections to be substantial. The fourth one depends on the third and it is difficult to predict how complicated it will be.

Acknowledgements.

We thank Mark Goresky, Helmut Hofer, Gopal Prasad and Bas Spitters for discussions.

AP was partly supported by NSF Grant DMS-0635607, an NSF CAREER Award DMS-1055897, Spain Ministry of Science Grant MTM 2010-21186-C02-01, and Spain Ministry of Science Sev-2011-0087. MAW is supported by the Oswald Veblen fund and also received support from NSF Grant DMS-0635607 during the preparation of this paper. VV is supported by NSF Grant DMS-1100938. This material is based upon work supported by the National Science Foundation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Appendix: Coq code

Disclaimer: The libraries summarized and listed below are in preliminary form and are actively being improved and extended by the authors and others. As such, we advise interested readers to consult also with the most recent versions, which need not agree in form and content with the libraries described here.

The Coq code is included in full below for easy reference by the reader. We also expect to make it available on the webpages of the first and third authors. For easy reference, we include here a brief sketch of the contents of each of the files. *It is worth remarking that all of the files described here rely upon the second author’s Coq library.* For more on this library we refer the reader to the library itself and to the tutorial [15]. For quick reference, Figures 1 and 2 give the dependences of the second author’s library and the library associated with this paper, respectively.

Of the new files, the file `lemmas.v` contains a number of small lemmas which, such as basic facts about apartness relations, some lemmas on rings, *et cetera*, which are required by the other files. The file `fps.v` contains all of the material on formal power series. The construction of the Heyting field of fractions can be found in `frac.v`. The basic number theoretic results which we require are in `zmodp.v`. Finally, the construction of the p -adic numbers is given in `padics.v`.

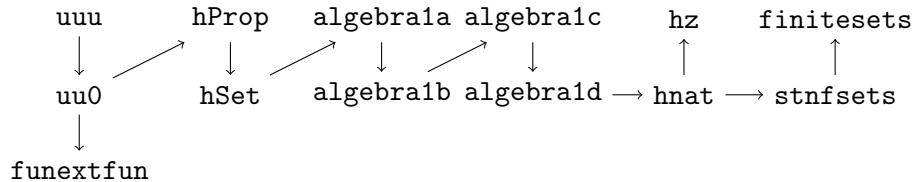


Figure 1: Dependence diagram of the second author’s Coq library

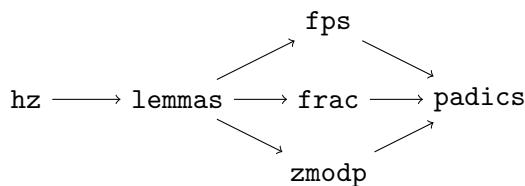


Figure 2: Dependence diagram of the additional Coq files for the p -adics

7.3 The file lemmas.v

```

(** *Fixing notation, terminology and basic lemmas *)
(** By Alvaro Pelayo, Vladimir Voevodsky and Michael A. Warren *)
(** Settings *)

Add Rec LoadPath "../Generalities". Add Rec LoadPath "../hlevel1".
Add Rec LoadPath "../hlevel2".

Unset Automatic Introduction. (** This line has to be removed for the
file to compile with Coq8.2 *)
(** Imports *)
Require Export hSet.
Require Export hnats.
Require Export Export hz.
(*Require Export finitesets.*)
Require Export stnfssets.*)

(** Fixing some notation *)
(** * Notation, terminology and very basic facts *)
Notation "x ~> y" := ( paths x y ) ( at level 50 ) : type_scope.

Implicit Arguments tpair [ T P ].

Lemma pathintotalfiber { B : UU } { E : B -> UU } { b0 b1 : B } { e0 :
E b0 } { e1 : E b1 } { p0 : b0 ~> b1 } { p1 : transportf E p0 e0 ~> e1 }
: (tpair b0 e0) ~> (tpair b1 e1). Proof. intros. destruct p0,
p1. apply idpath. Defined.

Definition neq { X : UU } : X -> X -> hProp := fun x y : X =>
hProppair (neg (x ~> y)) (isapropneg (x ~> y)).

Definition pathintotalpri { B : UU } { E : B -> UU } { v w : total2 E }
{ p : v ~> w } : (pri v) ~> (pri w) := maponpaths (fun x => pri
x) p.

Lemma isinclisinj { A B : UU } { f : A -> B } { p : isincl f } { a b :
A } { p : f a ~> f b } : a ~> b. Proof. intros. set (q := p (f a)).
set (a' := hfiberpair f a (idpath (f a))). set (b' :=
hfiberpair f b (pathsinv0 p0)). assert (a' ~> b') as p1. apply (p (f a)).
apply (pathintotalpri p1). Defined.

(** * I. Lemmas on natural numbers *)
Lemma minusOr { n : nat } : minus n 0 ~> n. Proof. intros
n. destruct n. apply idpath. apply idpath. Defined.

Lemma minusNn0 { n : nat } : minus n n ~> 0%nat. Proof.
intro. induction n. apply idpath. assumption. Defined.

Lemma minussn1 { n : nat } : minus (S n) 1 ~> n. Proof.
intro. destruct n. apply idpath. apply idpath. Defined.

Lemma minussnnon0 { n : nat } { p : natlh 0 n } : S (minus n 1) ~>
n. Proof. intro. destruct n. intro p. assert empty. exact (
isirreflnatlh 0%nat p). contradiction. intro. apply
maponpaths. apply minusOr. Defined.

Lemma minusleh { n m : nat } : natleh (minus n m) n. Proof. intros
n. induction n. intros m. apply isreflnatleh. intros m. destruct
m. apply isreflnatleh. exact (istransnatleh (minus n m) n (S n) (IHn m) (natlthtole n (S n) (natlthsn n))). Defined.

Lemma minusleh { n m : nat } { p : natlh 0 m } { q : natlh 0 m } { r :
natleh n m } : natleh (minus n 1) (minus m 1). Proof. intro
n. destruct n. auto. intros m p q r. destruct m. assert empty. exact (
isirreflnatlh 0%nat q). contradiction. assert (natleh n m) as
a. apply r. assert (natleh (minus n 0%nat) m) as a0. exact (
transportf (fun x : nat => natleh x m) (pathsinv0 (minus0 n)) a
). exact (transportf (fun x : nat => natleh (minus n 0) x) (
pathsinv0 (minus0 m)) a0). Defined.

Lemma minuslth { n m : nat } { p : natlh 0 n } { q : natlh 0 m } :
natlth (minus n m) n. Proof. intro n. destruct n. auto. intros m p
q. destruct m. assert empty. exact (isirreflnatlh 0%nat q
). contradiction. apply (natlehlthtrans _ n _). apply (minusleh n m
). apply natlthsn. Defined.

Lemma natlthsntoleh { n m : nat } : natlth m (S n) -> natleh m n.
Proof. intro. induction n. intros m p. destruct m. apply
isreflnatleh. assert (natlth m 0) as q. apply p. intro. unfold
natlth in q. exact (negnatlh0 m q). intros m p. destruct m. apply
natlehOn. apply (IHn m). assumption. Defined.

Lemma natlthminus { n m : nat } { p : natlh m n } : natlth 0 (minus
n m ). Proof. intro n. induction n. intros m p. assert empty. exact
(negnatlh0 m p). contradiction. intros m p. destruct
m. auto. apply IHn. apply p. Defined.

Lemma natlthsmminus { n m : nat } { p : natlh m n } : natlth
(minus (S n) (S m)) (S n). Proof. intro. induction n. intros m
p. assert empty. apply
nopathsfalse. assumption. contradiction. intros m p. destruct
m. assert (minus (S (S n)) 1 ~> S n) as f. destruct
n. auto. auto. rewrite f. apply natlthsn. apply (istransnatlh _ (S
n) _). apply IHn. assumption. apply natlthsn. Defined.

Lemma natlehsnminussms { n m : nat } { p : natleh m n } : natleh
(minus (S n) (S m)) (S n). Proof. intro n. induction n. intros m
p X. assert empty. apply nopathsfalse. assumption.
assumption. intros m p. destruct m. apply natlthtole. apply
natlthsn. apply (istransnatleh _ (S n) _). apply
IHn. assumption. apply natlthtole. apply natlthsn. Defined.

Lemma pathssminus { n m : nat } { p : natlh m (S n) } : S (minus n
m) ~> minus (S n) m. Proof. intro n. induction n. intros m
p. destruct m. auto. assert empty. apply nopathsfalse. apply
pathsinv0. assumption. contradiction. intros m p. destruct
m. auto. apply IHn. apply p. Defined.

Lemma natlehsminus { n m : nat } : natleh (minus (S n) m) (S (minus n
m)). Proof. intro n. induction n. intros m X. apply
nopathsfalse. apply pathsinv0. destruct
m. assumption. assumption. intros m. destruct m. apply
isreflnatleh. apply IHn. Defined.

```

```
Lemma natlthssminus { n m l : nat } ( p : natlth m ( S n ) ) ( q :
natlth l ( S ( minus ( S n ) m ) ) ) : natlth l ( S ( S n ) ). Proof.
intro n. intros m l p q. apply ( natlthletrans _ ( S ( minus ( S n )
m ) ) ). assumption. destruct m. apply isreflnatleh. apply
natlthtoleh. apply natlthsnminusmsn. assumption. Defined.
```

```
Lemma natdoubleminus { n k l : nat } ( p : natleh k n ) ( q : natleh l
k ) : ( minus n k ) ~> ( minus ( minus n l ) ( minus k l ) ). Proof.
intro n. induction n. auto. intros k l p q. destruct k. destruct l.
auto. assert empty. exact ( negnatlehsn0 l q ). contradiction.
destruct l. auto. apply ( IHn k l ). assumption. assumption. Defined.
```

```
Lemma minusnlehi1 { n m : nat } ( p : natlth m n ) : natleh m ( minus n
1 ). Proof. intro n. destruct n. intros m p. assert empty. exact ( negnatlthn0 m p ). contradiction. intros m p. destruct m. apply
natlehsn. apply natlthsnoleh. change ( minus ( S n ) 1 ) with ( minus
n 0 ). rewrite minusOr. assumption. Defined.
```

```
Lemma doubleminuslehpaths { n m : nat } ( p : natleh m n ) : minus n
(minus n m ) ~> m . Proof. intro n. induction n. intros m p. destruct
( natlechoice m 0 p ) as [ h | k ]. assert empty. apply negnatlthn0
with ( n := m ). assumption. contradiction. simpl. apply
pathsinv0. assumption.
```

```
intros. destruct m. simpl. apply minusnn0. change ( minus ( S n ) (
minus n m ) ~> S m ). rewrite <- pathssminus. rewrite IHn. apply
idpath. assumption. apply ( minuslth ( S n ) ( S m ) ). apply
natlthletrans _ n ). apply natlehsn. apply natlthletrans _ m ). apply natlehsn. Defined.
```

```
Lemma boolnegtrueimplfalse { v : bool } ( p : neg ( v ~> true ) ) : v
~> false. Proof. intros. destruct v. assert empty. apply
p. auto. contradiction. auto. Defined.
```

```
Definition natcoface { i : nat } : nat -> nat. Proof. intros i
n. destruct ( natgtb i n ). exact n. exact ( S n ). Defined.
```

```
Lemma natcofaceleh { i n upper : nat } ( p : natleh n upper ) : natleh
( natcoface i n ) ( S upper ). Proof. intros. unfold
natcoface. destruct ( natgtb i n ). apply natlthtoleh. apply
natlehtletrans _ upper ). assumption. apply natlthnsn. apply p.
Defined.
```

```
Lemma natgehimplnatgtbfalset { m n : nat } ( p : natgeh n m ) : natgtb
m n ~> false. Proof. intros. unfold natgeh in p. unfold natgh
p. apply boolnegtrueimplfalse. intro q. apply p. auto. Defined.
```

```
Definition natcofaceretract { i : nat } : nat -> nat. Proof. intros
i n. destruct ( natgtb i n ). exact n. exact ( minus n 1 ). Defined.
```

```
Lemma natcofaceretractisretract { i : nat } : funcomp ( natcoface i )
( natcofaceretract i ) ~> idfun nat. Proof. intro i. apply
funextfun. intro n. unfold funcomp. set ( c := natlthoregh n i
). destruct c as [ h | k ]. unfold natcoface. rewrite h. unfold
natcofaceretract. rewrite h. apply idpath. assert ( natgtb i n ~>
false ) as f. apply natgehimplnatgtbfalset. assumption. unfold
natcoface. rewrite f. unfold natcofaceretract. assert ( natgtb i ( S
n ) ~> false ) as ff. apply natgehimplnatgtbfalset. apply
istransnatgeh _ n ). apply natgtthtogh. apply natgthsnn. assumption.
rewrite ff. rewrite minusn1. apply idpath. Defined.
```

```
Lemma isinjnatcoface { i x y : nat } : natcoface i x ~> natcoface i y
-> x ~> y. Proof. intros i x y p. change x with ( ( idfun _ ) x
). rewrite <- ( natcofaceretractisretract i ). change y with ( ( idfun
_ ) y ). rewrite <- ( natcofaceretractisretract i ). unfold funcomp.
rewrite p. apply idpath. Defined.
```

```
Lemma natlehddecomp { b a : nat } : hexists ( fun c : nat => ( a + c
)%nat ~> b ) -> natleh a b. Proof. intro b. induction b. intros a
p. apply p. intro t. destruct t as [ c f ]. destruct a. apply
isreflnatleh. assert empty. simpl in f . exact ( negpathssx0 ( a + c
) f ). contradiction. intros a p. apply p. intro t. destruct t as [ c
f ]. destruct a. apply natlehsn. assert ( natleh a b ) as q. simpl in
f . apply IHb. intro P. intro s. apply s. split with c. apply
invmaponpathsS. assumption. apply q. Defined.
```

```
Lemma natdivleh { a b k : nat } ( f : ( a * k )%nat ~> b ) : coprod (
natleh a b ) ( b ~> 0%nat ). Proof. intros. destruct k. rewrite
natmultcomm in f. simpl in f. apply ii2. apply
pathsinv0. assumption. rewrite natmultcomm in f. simpl in f. apply
ii1. apply natlehddecomp. intro P. intro g. apply g. split with ( k * a
)%nat. assumption. Defined.
```

(** * II. Lemmas on rings *)

Open Scope rng_scope.

```
Lemma rngminusdistr { X : commrng } ( a b c : X ) : a * ( b - c ) ~> ( a
* b - a * c ). Proof. intros. rewrite rngldistr. rewrite
rngmultminus. apply idpath. Defined.
```

```
Lemma rngminusdistl { X : commrng } ( a b c : X ) : ( b - c ) * a ~> ( b
* a - c * a ). Proof. intros. rewrite rngrdistr. rewrite
rnglmultminus. apply idpath. Defined.
```

```
Lemma multinvmultstable { A : commrng } ( a b : A ) ( p : multinvpair
A a ) ( q : multinvpair A b ) : multinvpair A ( a * b ). Proof.
intros. destruct p as [ a' p ]. destruct q as [ b' q ]. split with ( b'
* a' ). split. change ( ( ( b' * a' ) * ( a * b ) )%rng ~> ( @runguel2 A ) ). rewrite ( rngassoc2 A b' ). rewrite <- ( rngassoc2 A
a' ). change ( dirprod ( ( a' * a )%rng ~> ( @runguel2 A ) ) ( ( a *
a' )%rng ~> ( @runguel2 A ) ) ) in p. change ( dirprod ( ( b' * b
)%rng ~> ( @runguel2 A ) ) ( ( b * b' )%rng ~> ( @runguel2 A ) ) ) in
q. rewrite <- ( pr1 q ). apply maponpaths. assert ( a' * a * b ~> 1 *
b ) as f. apply ( maponpaths ( fun x => x * b ) ( pr1 p ) ). rewrite
rnglunax2 in f. assumption. change ( ( ( a * b ) * ( b' * a' ) )%rng
~> ( @runguel2 A ) ). rewrite ( rngassoc2 A a ). rewrite <- ( rngassoc2
A b ). change ( dirprod ( ( a' * a )%rng ~> ( @runguel2 A ) ) ( ( a *
a' )%rng ~> ( @runguel2 A ) ) ) in p. change ( dirprod ( ( b' * b
)%rng ~> ( @runguel2 A ) ) ( ( b * b' )%rng ~> ( @runguel2 A ) ) ) in
q. rewrite <- ( pr2 q ). rewrite ( pr2 q ). rewrite rnglunax2. apply
p. Defined.
```

```
Lemma commrungaddinvunique { X : commrng } ( a b c : X ) ( p : @op1 X a
b ~> @runguel1 X ) ( q : @op1 X a c ~> @runguel1 X ) : b ~> c . Proof.
intros. rewrite ( pathsinv0 ( rngrunax1 X b ) ). rewrite ( pathsinv0
q ). rewrite ( pathsinv0 ( rngassoc1 X _ _ _ ) ). rewrite ( rngcommi
X b _ ). rewrite p. rewrite rnlunax1. apply idpath. Defined.
```

```
Lemma isapropmultinvpair { X : commrng } ( a : X ) : isaprop (
multinvpair X a ). Proof. intros. unfold isaprop. intros b c.
```

```
assert ( b ~> c ) as f. destruct b as [ b' b ]. destruct c as [ c' c
]. assert ( b ~> c ) as f0. rewrite <- ( @rungrunax2 X b ). change ( b
* ( @runguel2 X ) ) with ( b * 1 )%multmonoid. rewrite <- ( pr2 c
). change ( ( b * ( a * c ) )%rng ~> c ). rewrite <- ( rngassoc2 X
). change ( b * a )%rng with ( b * a )%multmonoid. rewrite ( pr1 b
). change ( ( @runguel2 X ) * c ~> c )%rng. apply rnlunax2. apply
pathintotalfiber with ( p0 := f0 ). assert ( isaprop ( dirprod ( c *
a ~> ( @runguel2 X ) ) ( a * c ~> ( @runguel2 X ) ) ) ) as is. apply
isofhleveldirprod. apply ( setproperty X ). apply ( setproperty X
). apply is. split with f. intros g. assert ( isaset ( multinvpair
```

X a)) as is. unfold multinvpair. unfold invpair. change isaset with (isofhlevel 2). apply isofhleveltotal2. apply (pr1 (pr1 (rigmultmonoid X))). intros. apply isofhlevekdirprod. apply hlevelntosn. apply (setproperty X). apply hlevelntosn. apply (setproperty X). apply is. Defined.

Close Scope rng_scope.

(** * III. Lemmas on hz *)

Open Scope hz _scope.

Lemma $hzaddinvplus$ ($n m : hz$) : $- (n + m) \simgt (-(n + m))$. Proof. intros. apply commrgaddinvunq with (a := n + m). apply rngrinvax1. assert ($(n + m) + (-n + -m) \simgt (n + -n + m + -m)$) as i. assert ($n + m + (-n + -m) \simgt (n + (m + (-n + -m)))$) as i0. apply rngrassoc1. assert ($n + (m + (-n + -m)) \simgt (n + (m + -n + -m))$) as i1. apply maponpaths. apply pathsinv0. apply rngrassoc1. assert ($n + (m + -n + -m) \simgt (n + (-n + m + -m))$) as i2. apply maponpaths. apply (maponpaths (fun x : _ => x + -m)). apply rngrcomm1. assert ($n + (-n + m + -m) \simgt (n + (-n + m) + -m)$) as i3. apply pathsinv0. apply rngrassoc1. assert ($n + (-n + m) + -m \simgt (n + -n + m + -m)$) as i4. apply pathsinv0. apply (maponpaths (fun x : _ => x + -m)). apply rngrassoc1. exact (pathscomp0 io (pathscomp0 ii (pathscomp0 i2 (pathscomp0 i3 i4)))). assert ($n + -n + m + -m \simgt 0$) as j. assert ($n + -n + m + -m \simgt (0 + m + -m)$) as j0. apply (maponpaths (fun x : _ => x + m + -m)). apply rngrinvax1. assert ($0 + m + -m \simgt (m + -m)$) as j1. apply (maponpaths (fun x : _ => x + -m)). apply rngrlunax1. assert ($m + -m \simgt 0$) as j2. apply rngrinvax1. exact (pathscomp0 jo (pathscomp0 ji j2)). exact (pathscomp0 i j). Defined.

Lemma $hzgthsntogeh$ ($n m : hz$) ($p : hzgth (n + 1) m$) : $hzgeh n m$. Proof. intros. set (c := hzgthchoice2 (n + 1) m). destruct c as [h | k]. exact p. assert (hzgth n m) as a. exact (hzgthandplusrinv n m 1 h). apply hzgthtoge. exact a. rewrite (hzplusrcan n m 1 k). apply isreflhzgeh. Defined.

Lemma $hzlthsntoleh$ ($n m : hz$) ($p : hzlth m (n + 1)$) : $hzleh m n$. Proof. intros. unfold hzlth in p. assert (hzgeh n m) as a. apply hzgthsntogeh. exact p. exact a. Defined.

Lemma $hzabsvalchoice$ ($n : hz$) : coprod ($0\%nat \simgt (hzabsval n)$) (total2 (fun x : nat => S x \simgt (hzabsval n))). Proof. intros. destruct (natlehchoice _ _ (natleh0n (hzabsval n))) as [l | r]. apply ii2. split (minus (hzabsval n) 1). rewrite pathsminus. change (minus (hzabsval n) 0) \simgt (hzabsval n). rewrite minusOr. apply idpath. assumption. apply iii1. assumption. Defined.

Lemma $hzlthminusswap$ ($n m : hz$) ($p : hzlth n m$) : $hzlth (-m) (-n)$. Proof. intros. rewrite <- (hzplusl0 (-m)). rewrite <- (hzminus n). change (hzlth (n + -n + -m) (-n)). rewrite hzplusassoc. rewrite (hzpluscomm (-n)). rewrite <- hzplusassoc. assert ($-n \simgt (0 + -n)$) as f. apply pathsinv0. apply hzplusl0. assert (hzlth (n + -m + -n) (0 + -n)) as q. apply hzlthandplusr. rewrite <- (hzminus m). change (m - m) with (m + -m). apply hzlthandplusr. assumption. rewrite <- f in q. assumption. Defined.

Lemma $hzlthminusequiv$ ($n m : hz$) : dirprod ((hzlth n m) \rightarrow (hzlth 0 (m - n))) ((hzlth 0 (m - n)) \rightarrow (hzlth n m)). Proof. intros. rewrite <- (hzminus n). change (n - n) with (n + -n). change (m - n) with (m + -n). split. intro p. apply hzlthandplusr. assumption. intro p. rewrite <- (hzplusl0).

).

rewrite <- (hzplusr0 m). rewrite <- (hzminus n). rewrite <- 2! hzplusassoc. apply hzlthandplusr. assumption. Defined.

Lemma $hzlthminus$ ($n m k : hz$) ($p : hzlth n k$) ($q : hzlth m k$) ($q' : hzleh 0 m$) : $hzlth (n - m) k$. Proof. intros. destruct (hzlehchoice 0 m q') as [l | r]. apply (istranshzlth _ n _). assert (hzlth (n - m) (n + 0)) as 10. rewrite <- (hzminus m). change (m - m) with (m + -m). rewrite <- (hzplusassoc). apply hzlthandplusr. assert (hzlth (n + 0) (n + m)) as 100. apply hzlthandplusl. assumption. rewrite (hzplusl0) in 100. assumption. rewrite hzplusr0 in 10. assumption. assumption. rewrite <- r. change (n - 0) with (n + -0). rewrite hzminuszero. rewrite (hzplusr0 n). assumption. Defined.

Lemma $hzabsvalandminuspos$ ($n m : hz$) ($p : hzleh 0 n$) ($q : hzleh 0 m$) : $hzttohz (hzabsval (n - m)) \simgt hzttohz (hzabsval (m - n))$. Proof. intros. destruct (hzlhorgeh n m) as [l | r]. assert (hzlth (n - m) 0) as a. change (n - m) with (n + -m). rewrite <- (hzminus m). change (m - m) with (m + -m). apply (hzlhandplusr). assumption. assert (hzlth 0 (m - n)) as b. change (m - n) with (m + -n). rewrite <- (hzminus n). change (n - n) with (n + -n). apply hzlhandplusr. assumption. rewrite (hzabsvalh0 a). rewrite hzabsvalgh0. change (n - m) with (n + -m). rewrite hzaddinvplus. change (-m) with (- -m) %rng. rewrite (rngrminusminus). rewrite hzpluscomm. apply idpath. apply b. destruct (hzgehchoice n m r) as [h | k]. assert (hzlth 0 (n - m)) as a. change (n - m) with (n + -m). rewrite <- (hzminus m). change (m - m) with (m + -m). apply hzlhandplusr. assumption. assert (hzlth (m - n) 0) as b. change (m - n) with (m + -n). rewrite <- (hzminus n). apply hzlhandplusr. apply h. rewrite (hzabsvalh0 b). rewrite (hzabsvalgh0). change (n + -m) \simgt - (m + -n). rewrite hzaddinvplus. change (-n) with (- -n) %rng. rewrite rngrminusminus. rewrite hzpluscomm. apply idpath. apply a. rewrite k. apply idpath. Defined.

Lemma $hzabsvalneq0$ ($n : hz$) ($p : hnq 0 n$) : $hzlth 0 (hzttohz (hzabsval n))$. Proof. intros. destruct (hnqchoice 0 n p) as [left | right]. rewrite hzabsvalh0. apply hzlh0andminus. apply left. apply left. rewrite hzabsvalgh0. assumption. apply right. Defined.

Definition $hzrdistr$ ($a b c : hz$) : $(a + b) * c \simgt ((a * c) + (b * c))$:= rngrdistr $hz a b c$.

Definition $hzldistr$ ($a b c : hz$) : $c * (a + b) \simgt ((c * a) + (c * b))$:= rnqlidistr $hz a b c$.

Lemma $hzabsvaland1$: $hzabsval 1 \simgt 1\%nat$. Proof. apply (isinclsinj isinclnattohz). rewrite hzabsvalgh0. rewrite nattohzand1. apply idpath. rewrite <- (hzplusl0 1). apply (hzlthnsn 0). Defined.

Lemma $hzabsvalandplusneg$ ($n m : hz$) ($p : hzleh 0 n$) ($q : hzleh 0 m$) : $hzabsval (n + m) \simgt ((hzabsval n) + (hzabsval m))\%nat$. Proof. intros. assert (hzleh 0 (n + m)) as r. rewrite <- (hzminus m). change (n - n) with (n + -n). apply hzlhandplusl. apply (istranshzleh _ 0 _). apply hzgeh0andminus. apply p. assumption. apply (isinclsinj isinclnattohz). rewrite nattohzandplus. rewrite 3!. rewrite hzabsvalgh0. apply idpath. apply q. apply p. apply r. Defined.

Lemma $hzabsvalandplusneg$ ($n m : hz$) ($p : hzlth n 0$) ($q : hzlth m 0$) : $hzabsval (n + m) \simgt ((hzabsval n) + (hzabsval m))\%nat$. Proof. intros. assert (hzlth (n + m) 0) as r. rewrite <- (

```

hzrminus n ). change ( n - n ) with ( n + - n ). apply
hzlthandplusl. apply (istranshzlth _ 0 _ ). assumption. apply
hzlthOandminus. assumption. apply (isinclisinj isinclnatth0).
rewrite nattohzandplus. rewrite 3! hzabsvalth0. rewrite
hzaddinvplus. apply idpath. apply q. apply p. apply r. Defined.

Lemma hzabsvalandnatth0 ( n : nat ) : hzabsval ( nattohz n ) ~> n.
Proof. induction n. rewrite nattohzand0. rewrite hzabsval0. apply
idpath. rewrite nattohzandS. rewrite hzabsvalandplusnonneg. rewrite
hzabsvalandi. simpl. apply maponpaths. assumption. rewrite <- (hzplus10 1). apply hzlthtoleh. apply (hzlthsn 0). rewrite <-
nattohzand0. apply nattohzandleh. apply natlehOn. Defined.

Lemma hzabsvalandth ( n m : hz ) ( p : hzleb 0 n ) ( p' : hzleb n m )
: natlth ( hzabsval n ) ( hzabsval m ). Proof. intros. destruct (
natlhgeh ( hzabsval n ) ( hzabsval m )) as [ h | k ].
assumption. assert empty. apply (isirreflhzlh m). apply (
hzlehlthtrans _ n _ ). rewrite <- (hzabsvalgeh0). rewrite <- (hzabsvalgeh p). apply nattohzandleh. apply k. apply
hzgthtoge. apply (hzgthgehtrans _ n _ ). apply p'. apply
p. assumption. contradiction. Defined.

Lemma nattohzandlthinv ( n m : nat ) ( p : hzleb ( nattohz n )
(nattohz m ) ) : natlth n m . Proof. intros. rewrite <- (hzabsvalandnatth0 n). rewrite <- (hzabsvalandnatth0 m). apply
hzabsvalandth. change 0 with (nattohz 0%nat). apply nattohzandleh.
apply natlehOn. assumption. Defined.

Close Scope hz_scope.

(** * IV. Generalities on apartness relations *)

```

Definition iscomparel { X : UU0 } (R : hrel X) := forall x y z : X,
R x y -> coprod (R x z) (R z y).

Definition isapart { X : UU0 } (R : hrel X) := dirprod (isirrefl R)
(dirprod (issymm R) (iscotrans R)).

Definition istightapart { X : UU0 } (R : hrel X) := dirprod (isapart R)
(forall x y : X, neg (R x y) -> (x ~> y)).

Definition apart (X : hSet) := total2 (fun R : hrel X => isapart R
).

Definition isbinopapartl { X : hSet } (R : apart X) (opp : binop X)
:= forall a b c : X, ((pr1 R) (opp a b) (opp a c)) -> (pr1
R) b c.

Definition isbinopapapr { X : hSet } (R : apart X) (opp : binop X)
:= forall a b c : X, (pr1 R) (opp b a) (opp c a) -> (pr1
R) b c.

Definition isbinopapart { X : hSet } (R : apart X) (opp : binop X)
:= dirprod (isbinopapartl R opp) (isbinopapapr R opp).

Lemma deceqtoeqpart { X : UU0 } (is : isdeceq X) : isapart (neq X
). Proof. intros. split. intros a. intro p. apply p. apply idpath.
split. intros a b p q. apply p. apply pathsinv0. assumption. intros a
c b p s. apply s. destruct (is a c) as [l | r]. apply
ii2. rewrite <- l. assumption. apply ii1. assumption. Defined.

Definition isapartdec { X : hSet } (R : apart X) := forall a b : X,
coprod ((pr1 R) a b) (a ~> b).

Lemma isapartdectodeceq { X : hSet } (R : apart X) (is : isapartdec
R) : isdeceq X. Proof. intros X R is y z. destruct (is y z) as [

```

l | r ]. apply ii2. intros f. apply ( ( pr1 ( pr2 R ) ) z ). rewrite f
in l. assumption. apply ii1. assumption. Defined.

Lemma isdeceqtoisapartdec ( X : hSet ) ( is : isdeceq X ) : isapartdec
( tpair _ ( deceqtoeqpart is ) ). Proof. intros X is a b. destruct
( is a b ) as [ l | r ]. apply ii2. assumption. apply ii1. intros
f. apply r. assumption. Defined.

(** * V. Apartness relations on rings *)

Open Scope rng_scope.

Definition acommrng := total2 ( fun X : commrng => total2 ( fun R :
apart X => dirprod ( isbinopapart R ( @op1 X ) ) ( isbinopapart R ( @op2 X ) ) ) ).

Definition acommrngpair := tpair ( P := fun X : commrng => total2 ( fun R : apart X => dirprod ( isbinopapart R ( @op1 X ) ) ( isbinopapart R ( @op2 X ) ) ) ). Definition acommrngconstr := acommrngpair.

Definition acommrngtocommrng : acommrng -> commrng := @pr1 _ _.
Coercion acommrngtocommrng : acommrng -> commrng.

Definition acommrngapartrel ( X : acommrng ) := pr1 ( pr1 ( pr2 X ) ).
Notation " a # b " := ( acommrngapartrel _ a b ) ( at level 50 ) :
rng_scope.

Definition acommrng_aadd ( X : acommrng ) : isbinopapart ( pr1 ( pr2 X
) ) op1 := ( pr1 ( pr2 ( pr2 X ) ) ). Definition acommrng_amult ( X :
acommrng ) : isbinopapart ( pr1 ( pr2 X ) ) op2 := ( pr2 ( pr2 ( pr2 X
) ) ). Definition acommrng_airefl ( X : acommrng ) : isirrefl ( pr1
( pr1 ( pr2 X ) ) ) := pr1 ( pr2 ( pr1 ( pr2 X ) ) ). Definition
acommrng_asymm ( X : acommrng ) : issymm ( pr1 ( pr1 ( pr2 X ) ) ) := pr1
( pr2 ( pr2 ( pr1 ( pr2 X ) ) ) ). Definition acommrng_acotrans ( X :
acommrng ) : iscotrans ( pr1 ( pr1 ( pr2 X ) ) ) := pr2 ( pr2
( pr1 ( pr2 X ) ) ) .

Definition aintdom := total2 ( fun A : acommrng => dirprod ( ( rngunel2 ( X := A ) ) # 0 ) ( forall a b : A, ( a # 0 ) -> ( b # 0 )
-> ( ( a * b ) # 0 ) ) ).

Definition aintdompair := tpair ( P := fun A : acommrng => dirprod ( ( rngunel2 ( X := A ) ) # 0 ) ( forall a b : A, ( a # 0 ) -> ( b # 0 )
-> ( ( a * b ) # 0 ) ) ). Definition aintdomconstr := aintdompair.

Definition priaintdom : aintdom -> acommrng := @pr1 _ _.
Coercion priaintdom : aintdom -> acommrng.

Definition aintdom := total2 ( fun A : acommrng => dirprod ( ( rngunel2 ( X := A ) ) # 0 ) ( forall a b : A, ( a # 0 ) -> ( b # 0 )
-> ( ( a * b ) # 0 ) ) ). Definition aintdomconstr := aintdompair.

Definition priaintdom : aintdom -> acommrng := @pr1 _ _.
Coercion priaintdom : aintdom -> acommrng.

Definition isaafield ( A : acommrng ) := dirprod ( ( rngunel2 ( X := A
) ) # 0 ) ( forall x : A, x # 0 -> multinvpair A x ).

Definition afld := total2 ( fun A : acommrng => isaafield A ). Definition afldpair ( A : acommrng ) ( is : isaafield A ) : afld := tpair A is . Definition priafld : afld -> acommrng := @pr1 _ _ .
Coercion priafld : afld -> acommrng.

Lemma afldinvertibletozero ( A : afld ) ( a : A ) ( p : multinvpair A
a ) : a # 0. Proof. intros. destruct p as [ a' p ]. assert ( a' * a
# 0 ) as q. change ( a' * a # 0 ). assert ( a' * a ~> a * a' ) as
f. apply ( rngcomm2 A ). assert ( a * a' ~> 1 ) as g. apply
```

```

p. rewrite f, g. apply A. assert ( a' * a # a' * ( rnguneli ( X := A ) ) ) as q'. assert ( ( rnguneli ( X := A ) ) ^> ( a' * ( rnguneli ( X := A ) ) ) ) as f. apply pathsinv0. apply ( rngmultx0 A ). rewrite <- f. assumption. apply ( pr1 ( acommrng_amult A ) ) a'. assumption. Defined.

Definition afldtoaintdom ( A : afld ) : aintdom . Proof. intro . split with ( pr1 A ) . split. apply A. intros a b p q. apply afldinvertiblezero. apply multinvmultstable. apply A. assumption. apply A. assumption. Defined.

Lemma timesazer0 { A : acommrng } { a b : A } ( p : a * b # 0 ) : dirprod ( a # 0 ) ( b # 0 ). Proof. intros. split. assert ( a * b # 0 * b ) as h. rewrite ( rngmultx0 A ). assumption. apply ( pr2 ( acommrng_amult A ) b ). assumption. apply ( pr1 ( acommrng_amult A ) a ). rewrite ( rngmultx0 A ). assumption. Defined.

Lemma aaminuszero { A : acommrng } { a b : A } ( p : a # b ) : ( a - b ) # 0 . Proof. intros. rewrite <- ( rngrunax1 A a ) in p. rewrite <- ( rngrunax1 A b ) in p. assert ( a + 0 ^> ( a + ( b - b ) ) ) as f. rewrite <- ( rngrunvax1 A b ). apply idpath. rewrite f in p. rewrite <- ( rngmultwithminus1 A ) in p. rewrite <- ( rngassoc1 A ) in p. rewrite ( rngcomm1 A a ) in p. rewrite ( rngassoc1 A b ) in p. rewrite ( rngmultwithminus1 A ) in p. apply ( pr1 ( acommrng_aadd A ) ) b ( a - b ) 0 . assumption. Defined.

Lemma aminuszeroa { A : acommrng } { a b : A } ( p : ( a - b ) # 0 ) : a # b . Proof. intros. change 0 with ( @rnguneli A ) in p. rewrite <- ( rngrunvax1 A b ) in p. rewrite <- ( rngmultwithminus1 A ) in p. apply ( pr2 ( acommrng_aadd A ) ) ( -1 * b ) a b . assumption. Defined.

27 Close Scope rng_scope.

(** * VI. Lemmas on logic *)

Lemma horelim ( A B : UU ) ( P : hProp ) : dirprod ( isinh_UU A -> P ) ( isinh_UU B -> P ) -> ( hdij A B -> P ) . Proof. intros A B P p. intro q. intro u. destruct u as [ u | v ]. apply ( pr1 p ). intro Q. auto. apply ( pr2 p ). intro Q. auto. Defined.

Lemma stronginduction { E : nat -> UU } ( p : E 0%nat ) ( q : forall n : nat, natneq n 0%nat -> ( forall m : nat, natlh m n -> E m ) -> E n ) : forall n : nat, E n . Proof. intros. destruct n. assumption. apply q. apply ( negpathssx0 n ). induction n. intros m t. rewrite ( natithitois0 m t ). assumption. intros m t. destruct ( natlechoice _ _ ( natithsntoleh _ _ t ) ) as [ left | right ]. apply Ihn. assumption. apply q. rewrite right. intro f. apply ( negpathssx0 n ). assumption. intros k s. rewrite right in s. apply ( Ihn k ). assumption. Defined.

Lemma setquotpprpathsandR { X : UU } ( R : eqrel X ) : forall x y : X, setquotpr R x ^> setquotpr R y -> R x y . Proof. intros. assert ( pr1 ( setquotpr R x ) y ) as i. assert ( pr1 ( setquotpr R y ) x ) as i0. unfold setquotpr. apply R. destruct X0. assumption. apply i. Defined.

(* Some lemmas on decidable properties of natural numbers. *)
Definition isdecnatprop ( P : nat -> hProp ) := forall m : nat, coprod ( P m ) ( neg ( P m ) ).

Lemma negisdecnatprop ( P : nat -> hProp ) ( is : isdecnatprop P ) : isdecnatprop ( fun n : nat => hneg ( P n ) ). Proof. intros P is n. destruct ( is n ) as [ l | r ]. apply ii2. intro j. assert hfalse as x. apply j. assumption. apply x. apply iii1. assumption. Defined.

Lemma bndexistsisdecnatprop ( P : nat -> hProp ) ( is : isdecnatprop P ) : isdecnatprop ( fun n : nat => hexists ( fun m : nat => dirprod ( natleh m n ) ( P m ) ) ). Proof. intros P is n. induction n. destruct ( is 0%nat ) as [ l | r ]. apply iii1. apply total2tohexists. split with 0%nat. split. apply isreflnatleh. assumption. apply ii2. intro j. assert hfalse as x. apply j. intro m. destruct m as [ m m' ]. apply r. rewrite ( natleh0tois0 m ( pr1 m' ) ) in m'. apply m'. apply x.

destruct ( is ( S n ) ) as [ l | r ]. apply iii1. apply total2tohexists. split with ( S n ). split. apply ( isreflnatleh ( S n ) ). assumption. destruct Ihm as [ l' | r' ]. apply iii1. apply l'. intro m. destruct m as [ m m' ]. apply total2tohexists. split with m. split. apply ( istransnatleh _ n _ ). apply m'. apply natlthtoleh. apply natlthnsn. apply m'. apply ii2. intro j. apply r'. apply j. intro m. destruct m as [ m m' ]. apply total2tohexists. split with m. split. destruct ( natlechoice m ( S n ) ( pr1 m' ) ). apply natlthsntoleh. assumption. assert empty. apply r. rewrite <- i. apply m'. contradiction. apply m'. Defined.

Lemma isdecisbndqdec ( P : nat -> hProp ) ( is : isdecnatprop P ) ( n : nat ) : coprod ( forall m : nat, natleh m n -> P m ) ( hexists ( fun m : nat => dirprod ( natleh m n ) ( neg ( P m ) ) ) ). Proof. intros P is n. destruct ( bndexistisisdecnatprop _ ( negisdecnatprop P is ) n ) as [ l | r ]. apply ii2. assumption. apply iii1. intros m j. destruct ( is m ) as [ l' | r' ]. assumption. assert hfalse as x. apply r. apply total2tohexists. split with m. split. assumption. contradiction. Defined.

Lemma leastelementprinciple ( n : nat ) ( P : nat -> hProp ) ( is : isdecnatprop P ) : P n -> hexists ( fun k : nat => dirprod ( P k ) ( forall m : nat, natlh m k -> neg ( P m ) ) ) . Proof. intro n. induction n. intros P is u. apply total2tohexists. split with 0%nat. split. assumption. intros m i. assert empty. apply ( negnatgth0n m i ). contradiction. intros P is u. destruct ( is 0%nat ) as [ l | r ]. apply total2tohexists. split with 0%nat. split. assumption. intros m i. assert empty. apply ( negnatgth0n m i ). contradiction. set ( P' := fun m : nat => P ( S m ) ). assert ( forall m : nat, coprod ( P' m ) ( neg ( P' m ) ) ) as is'. intros m. unfold P'. apply ( is ( S m ) ). set ( c := Ihn P' is' u ). apply c. intros k. destruct k as [ k v ]. destruct v as [ v0 v1 ]. apply total2tohexists. split with ( S k ). split. assumption. intros m. destruct m. intros i. assumption. intros i. apply v1. apply i. Defined.

(** END OF FILE *)

```

7.4 The file fps.v

```

(** *Formal Power Series *)

(** By Alvaro Pelayo, Vladimir Voevodsky and Michael A. Warren *)

(** January 2011 *)

(** Settings *)

Add Rec LoadPath "../Generalities". Add Rec LoadPath "../hlevel1".
Add Rec LoadPath "../hlevel2". Add Rec LoadPath
"../Proof_of_Extensionality". Add Rec LoadPath "../Algebra".

Unset Automatic Introduction. (** This line has to be removed for the
file to compile with Coq8.2*)

(** Imports *)

Require Export lemmas.

(** ** I. Summation in a commutative ring *)

Open Scope rng_scope.

Definition natsummation0 {R : commrng} (upper : nat) (f : nat -> R) :
R. Proof. intro R. intro upper. induction upper. intros. exact
(f 0%nat). intros. exact ((IHupper f + (f (S upper))). Defined.

Lemma natsummationpaths {R : commrng} {upper upper' : nat} (u :
upper > upper') (f : nat -> R) : natsummation0 upper f >
natsummation0 upper' f. Proof. intros. destruct u. auto. Defined.

Lemma natsummationpathsupfixed {R : commrng} {upper : nat} (f f' :
nat -> R) (p : forall x : nat, natleh x upper -> f x > f' x) :
natsummation0 upper f > natsummation0 upper f'. Proof. intros R
upper. induction upper. intros f f' p. simpl. apply p. apply
isreflnatleh. intros. simpl. rewrite (IHupper f f'). rewrite (p (S upper)). apply idpath. apply isreflnatleh. intros x p'. apply
p. apply (istransnatleh _ upper). assumption. apply
natlthtoleh. apply natlthnsn. Defined.

(* Here we consider summation of functions which are, in a fixed
interval, 0 for all but either the first or last value. *)

Lemma natsummationaeobottom {R : commrng} {f : nat -> R} (upper :
nat) (p : forall x : nat, natlth 0 x -> f x > 0) : natsummation0
upper f > (f 0%nat). Proof. intros R f upper. induction
upper. auto. intro p. simpl. rewrite (IHupper). rewrite (p (S
upper)). rewrite (rngrunaxi R). apply idpath. apply
natlehltrans _ upper. apply natlehOn. apply
natlthnsn. assumption. Defined.

Lemma natsummationaeotop {R : commrng} {f : nat -> R} (upper :
nat) (p : forall x : nat, natlth x upper -> f x > 0) :
natsummation0 upper f > (f upper). Proof. intros R f
upper. induction upper. auto. intro p. assert (natsummation0 upper f
> (natsummation0 (R := R) upper (fun x : nat => 0))) as g.
apply natsummationpathsupfixed. intros m q. apply p. exact
(natlehltrans m upper (S upper) q (natlthnsn upper)).
simpl. rewrite g. assert (natsummation0 (R := R) upper (fun _ :
nat => 0) > 0) as g'. set (g' := fun x : nat => rnguneli (X := R
)). assert (forall x : nat, natlth 0 x -> g' x > 0) as q0. intro
k. intro pp. auto. exact (natsummationaeobottom upper q0). rewrite
g'. rewrite (rnglunaxi R). apply idpath. Defined.

Lemma natsummationshift0 {R : commrng} (upper : nat) (f : nat ->
R) : natsummation0 (S upper) f > (natsummation0 upper (fun x :
nat => f (S x)) + f 0%nat). Proof. intros R upper. induction
upper. intros f. simpl. apply R. intros. change (natsummation0 (S
upper) f + f (S (S upper))) > (natsummation0 upper (fun x : nat
=> f (S x)) + f (S (S upper)) + f 0%nat). rewrite
IHupper. rewrite 2! (rngassoc R). rewrite (rngcommi R (f 0%nat)
- ). apply idpath. Defined.

Lemma natsummationshift {R : commrng} (upper : nat) (f : nat -> R) {i : nat} (p : natleh i upper) : natsummation0 (S upper) f >
(natsummation0 upper (funcmp (natcoface i) f) + f i). Proof.
intros R upper. induction upper. intros f i p. destruct i. unfold
funcmp. apply R. assert empty. exact (negatelehsn0 i p
). contradiction. intros f i p. destruct i. apply natsummationshift0.
destruct (natlechoice (S i) (S upper) p) as [h | k]. change
(natsummation0 (S upper) f + f (S (S upper))) > (natsummation0
(S upper) (funcmp (natcoface (S i)) f) + f (S i))
. rewrite (IHupper f (S i)). simpl. unfold funcmp at 3. unfold
natcoface at 3. rewrite 2! (rngassoc R). rewrite (rngcommi R _ (f (S i))). simpl. rewrite (natgeimplnatgtbfalse i upper). apply
idpath. apply p. apply natlthnsn. assumption. simpl. assert (
natsummation0 upper (funcmp (natcoface (S i)) f) >
natsummation0 upper f) as h. apply
natsummationpathsupfixed. intros m q. unfold funcmp. unfold
natcoface. assert (natlth m (S i)) as q'. apply (natlehltrans -
upper). assumption. rewrite k. apply natlthnsn. unfold natlth in q'.
rewrite q'. apply idpath. rewrite <- h. unfold funcmp, natcoface at
3. simpl. rewrite (natgeimplnatgtbfalse i upper). rewrite 2! (rngassoc R). rewrite (rngcommi R (f (S (S upper)))). rewrite
k. apply idpath. apply p. Defined.

Lemma natsummationplusdistr {R : commrng} (upper : nat) (f g :
nat -> R) : natsummation0 upper (fun x : nat => f x + g x) > (((
natsummation0 upper f) + (natsummation0 upper g)). Proof. intros
R upper. induction upper. auto. intros f g. simpl. rewrite <- (rngassoc R _ (natsummation0 upper g) _). rewrite (rngassoc1 R (natsummation0
upper f)). rewrite <- (rngassoc1 R (natsummation0 upper f)). rewrite
<- (IHupper f g). rewrite (rngassoc R). apply idpath. Defined.

Lemma natsummationtimesdistr {R : commrng} (upper : nat) (f : nat
-> R) (k : R) : k * (natsummation0 upper f) > (natsummation0
upper (fun x : nat => k * f x)). Proof. intros R upper. induction
upper. auto. intros f k. simpl. rewrite <- (IHupper). rewrite <- (rngidistr R). apply idpath. Defined.

Lemma natsummationtimesdistl {R : commrng} (upper : nat) (f : nat
-> R) (k : R) : (natsummation0 upper f) * k > (natsummation0
upper (fun x : nat => f x * k)). Proof. intros R upper. induction
upper. auto. intros f k. simpl. rewrite <- IHupper. rewrite
(rngrdistr R). apply idpath. Defined.

Lemma natsummationsswapminus {R : commrng} {upper n : nat} (f :
nat -> R) (q : natleh n upper) : natsummation0 (S (minus upper n
)) f > natsummation0 (minus (S upper) n) f. Proof. intros R
upper. induction upper. intros n f q. destruct n. auto. assert
empty. exact (negatelehsn0 n q). contradiction. intros n f
q. destruct n. auto. change (natsummation0 (S (minus upper n)) f
> natsummation0 (minus (S upper) n) f). apply IHupper. apply q.
Defined.

(** The following lemma asserts that
$ \sum^n_{k=0} \sum^k_{l=0} f(l, k-l) = \sum^n_{k=0} \sum^{n-k}_{l=0} f(k, l) $ *)

```

```
Lemma natsumsummationswap { R : commrng } { upper : nat } { f : nat -> nat -> R } : natsumsummation0 upper ( fun i : nat -> natsumsummation0 i ( fun j : nat => f j ( minus i j ) ) ) ~> ( natsumsummation0 upper ( fun k : nat -> natsumsummation0 ( minus upper k ) ( fun l : nat => f k l ) ) ). Proof. intros R upper. induction upper. auto.
```

```
intros f. change ( natsumsummation0 upper ( fun i : nat -> natsumsummation0 i ( fun j : nat => f j ( minus i j ) ) ) + natsumsummation0 ( S upper ) ( fun j : nat -> f j ( minus ( S upper ) j ) ) ) ~> ( natsumsummation0 upper ( fun k : nat -> natsumsummation0 ( S upper - k ) ( fun l : nat => f k l ) ) + natsumsummation0 ( minus ( S upper ) ( S upper ) ) ( fun l : nat -> f ( S upper ) l ) ). change ( natsumsummation0 upper ( fun i : nat -> natsumsummation0 i ( fun j : nat => f j ( minus i j ) ) ) + ( natsumsummation0 upper ( fun j : nat => f j ( minus ( S upper ) j ) ) + f ( S upper ) ( minus ( S upper ) ( S upper ) ) ) ~> ( natsumsummation0 upper ( fun k : nat -> natsumsummation0 ( S upper - k ) ( fun l : nat => f k l ) ) + natsumsummation0 ( minus ( S upper ) ( S upper ) ) ( fun l : nat -> f ( S upper ) l ) ).
```

```
assert ( (natsumsummation0 upper ( fun k : nat -> natsumsummation0 ( S ( minus upper k ) ) ( fun l : nat => f k l ) ) ) ~> (natsumsummation0 upper ( fun k : nat -> natsumsummation0 ( minus ( S upper ) k ) ( fun l : nat => f k l ) ) ) ) as A. apply natsumsummationpathswapminus. exact q. rewrite <- A. change ( fun k : nat -> natsumsummation0 ( S ( minus upper k ) ) ( fun l : nat => f k l ) ) with ( fun k : nat -> natsumsummation0 ( minus upper k ) ( fun l : nat => f k l ) ) + f k ( S ( minus upper k ) ). rewrite ( natsumsummationplusdistr upper _ ( fun k : nat -> f k ( S ( minus upper k ) ) ) ). rewrite IHupper. rewrite minusnn0. rewrite ( rngassoc1 R). assert ( natsumsummation0 upper ( fun j : nat => f j ( minus ( S upper ) j ) ) ~> natsumsummation0 upper ( fun k : nat -> f k ( S ( minus upper k ) ) ) ) as g. apply natsumsummationpathswapminus. intros m q. rewrite pathssminus. apply idpath. apply ( natlehlthtrans _ upper ). assumption. apply natlthnsn. rewrite g. apply idpath. Defined.
```

(** * II. Reindexing along functions $i : \text{nat} \rightarrow \text{nat}$ which are automorphisms of the interval of summation.*)

```
Definition isnattruncauto ( upper : nat ) { i : nat -> nat } := dirprod ( forall x : nat, natleh x upper -> total2 ( fun y : nat => dirprod ( natleh y upper ) ( dirprod ( i y ~> x ) ( forall z : nat, natleh z upper -> i z ~> x -> y ~> z ) ) ) ) ( forall x : nat, natleh x upper -> natleh ( i x ) upper ).
```

```
Lemma nattruncautoisnj { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } { n m : nat } { n' : natleh n upper } { m' : natleh m upper } : i n ~> i m -> n ~> m. Proof. intros upper i p n m n' m'. assert ( natleh ( i m ) upper ) as q. apply p. assumption. set ( x := pr1 p ( i m ) q ). set ( v := pr1 i ( y ~> x ) ). set ( w := pr1 ( pr2 x ) ). set ( y := pr1 ( pr2 ( pr2 x ) ) ). change ( pr1 x ) with v in w, y. assert ( v ~> n ) as a. apply ( pr2 x ). assumption. assumption. rewrite <- a. apply ( pr2 x ). assumption. assumption. rewrite <- a. apply ( pr2 x ). assumption. apply idpath. Defined.
```

```
Definition nattruncautopreimage { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } { n : nat } { n' : natleh n upper } : nat := pr1 ( pr1 p n n' ).
```

```
Definition nattruncautopreimagepath { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } { n : nat } { n' : natleh n upper } : i ( nattruncautopreimage p n' ) ~> n := ( pr1 ( pr2 ( pr2 ( pr1 p n n' ) ) ) ).
```

```
Definition nattruncautopreimageineq { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } { n : nat } { n' : natleh n upper } :
```

```
natleh ( nattruncautopreimage p n' ) upper := ( ( pr1 ( pr2 ( pr1 p n n' ) ) ) ).
```

```
Definition nattruncautopreimagecanon { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } { n : nat } { n' : natleh n upper } { m : nat } { m' : natleh m upper } { q : i m ~> n } : nattruncautopreimage p n' ~> m := ( pr2 ( pr2 ( pr2 ( pr1 p n n' ) ) ) ) m m' q.
```

```
Definition nattruncautoinv { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } : nat -> nat. Proof. intros upper i p n. destruct ( natgthorle h n upper ) as [ l | r ]. exact n. exact ( nattruncautopreimage p r ). Defined.
```

```
Lemma nattruncautoinvsnattruncauto { upper : nat } { i : nat -> nat } { p : isnattruncauto upper i } : isnattruncauto upper ( nattruncautoinv p ). Proof. intros. split. intros n n'. split with ( i n ). split. apply p. assumption. split. unfold nattruncautoinv. destruct ( natgthorle ( i n ) upper ) as [ l | r ]. assert empty. apply ( isirreflnatlh ( i n ) ). apply ( natlehlthtrans _ upper ). apply p. assumption. assumption. contradiction. apply ( nattruncautoisnj p ). apply ( nattruncautopreimageineq ). assumption. apply ( nattruncautopreimagepath p r ). intros x v. unfold nattruncautoinv in v. destruct ( natgthorle m upper ) as [ l | r ]. assert empty. apply ( isirreflnatlh upper ). apply ( natlthletrans _ m ). assumption. assumption. contradiction. rewrite <- v. apply ( nattruncautopreimagepath p r ). intros x X. unfold nattruncautoinv. destruct ( natgthorle x upper ) as [ l | r ]. assumption. apply ( nattruncautopreimageineq p r ). Defined.
```

```
Definition truncautotruncnauto { upper : nat } { i : nat -> nat } { p : isnattruncauto ( S upper ) i } : nat -> nat. Proof. intros upper i p n. destruct ( natlthorgeh ( i n ) ( S upper ) ) as [ l | r ]. exact ( i n ). destruct ( natgchchoice _ r ) as [ a | b ]. exact ( i n ). destruct ( isdeceqnat n ( S upper ) ) as [ h | k ]. exact ( i n ). exact ( i ( S upper ) ). Defined.
```

```
Lemma truncautotruncnautobound { upper : nat } { i : nat -> nat } { p : isnattruncauto ( S upper ) i } { n : nat } { q : natleh n upper } : natleh ( truncautotruncnauto p n ) upper. Proof. intros. unfold truncautotruncnauto. destruct ( natlthorgeh ( i n ) ( S upper ) ) as [ l | r ]. apply natlthsntoleh. assumption. destruct ( natgchchoice ( i n ) ( S upper ) ) as [ l' | r' ]. assert empty. apply ( isirreflnatlh ( i n ) ). apply ( natlehlthtrans _ ( S upper ) ). apply p. apply natltholeh. apply ( natlehlthtrans _ upper ). assumption. apply natlthnsn. assumption. contradiction. destruct ( isdeceqnat ( S upper ) ) as [ l'' | r'' ]. assert empty. apply ( isirreflnatlh upper ). apply ( natlthletrans _ ( S upper ) ). apply natlthnsn. rewrite <- l''. assumption. contradiction. assert ( natleh ( i ( S upper ) ) ( S upper ) ) as aux. apply p. apply isreflnatleh. destruct ( natgchchoice _ aux ) as [ l''' | r''' ]. apply natlthsntoleh. assumption. assert empty. apply r''. apply ( nattruncautoisnj p ). apply natltholeh. apply ( natlehlthtrans _ upper ). assumption. apply natlthnsn. apply isreflnatleh. rewrite r''. rewrite r''' . apply idpath. contradiction. Defined.
```

```
Lemma truncautotruncnautoisnj { upper : nat } { i : nat -> nat } { p : isnattruncauto ( S upper ) i } { n m : nat } { n' : natleh n upper } { m' : natleh m upper } : truncautotruncnauto p n ~> truncautotruncnauto p m ~> n ~> m. Proof. intros upper i p n m q r s. apply ( nattruncautoisnj p ). apply natltholeh. apply ( natlehlthtrans _ upper ). assumption. apply natlthnsn. apply natltholeh. apply ( natlehlthtrans _ upper ). assumption. apply natlthnsn. unfold truncautotruncnauto in s. destruct ( natlthorgeh ( i n ) ( S upper ) ) as [ a0 | a1 ]. destruct ( natlthorgeh ( i m ) ( S upper ) ) as [ b0 |
```

```

b1 ]. assumption. assert empty. assert ( i m ~> S upper ) as f0.
destruct ( natgehchoice ( i m ) ( S upper ) b1 ) as [ l | l' ]. assert
empty. apply ( isirreflnatlh ( S upper ) ). apply ( natlehlthtrans -
( i n ) ). rewrite
s. assumption. assumption. contradiction. assumption. destruct
( natgehchoice ( i m ) ( S upper ) b1 ) as [ a00 | a10 ]. apply (
isirreflnatlh ( S upper ) ). rewrite f0 in a00. assumption. destruct
( isdeceqat m ( S upper ) ) as [ a00 | a100 ]. rewrite s in
a0. rewrite f0 in a0. apply ( isirreflnatlh ( S upper ) )
). assumption. assert ( i m ~> n ) as f1. apply ( natruncautoisnj p
). rewrite f0. apply isreflnatleh. apply natlthtoleh. apply (
natlehlthtrans _ upper ). assumption. apply natlthnsn. rewrite f0.
rewrite s. apply idpath. apply ( isirreflnatlh upper ). apply
natlthtrans _ n ). rewrite <- f1, f0. apply
natlthnsn. assumption. contradiction. destruct ( natgehchoice ( i n )
( S upper ) a1 ) as [ a00 | a01 ]. assert empty. apply (
isirreflnatlh ( S upper ) ). apply ( natlthlehrans - ( i n )
). assumption. apply ( p ). apply natlthtoleh. apply ( natlehlthtrans
_ upper ). assumption. apply natlthnsn. contradiction. destruct
( natlthorgeh ( i m ) ( S upper ) ) as [ b0 | b1 ]. destruct
( isdeceqat n ( S upper ) ) as [ a000 | a001 ]. assumption. assert ( S
upper ~> m ) as f0. apply ( natruncautoisnj p ). apply
isreflnatleh. apply natlthtoleh. apply ( natlehlthtrans _ upper
). assumption. apply natlthnsn. assumption. assert empty. apply
a001. rewrite f0. assert empty. apply ( isirreflnatlh ( S upper )
). apply ( natlehlthtrans _ upper ). rewrite f0. assumption. apply
natlthnsn. contradiction. contradiction. destruct ( natgehchoice ( i
m ) ( S upper ) b1 ) as [ b00 | b01 ]. assert empty. apply (
isirreflnatlh ( i m ) ). apply ( natlehlthtrans _ ( S upper )
). apply p. apply ( natlthtoleh ). apply ( natlehlthtrans _ upper
). assumption. apply natlthnsn. assumption. contradiction. rewrite
b01. rewrite a01. apply idpath. Defined.

```

```

Lemma truncnattruncautoisauto { upper : nat } { i : nat -> nat } { p :
isnattruncauto ( S upper ) i } : isnattruncauto upper
( truncnattruncauto p ). Proof. intros. split. intros n q. assert (
natleh n ( S upper ) ) as q'. apply natlthtoleh. apply (
natlehlthtrans _ upper ). assumption. apply natlthnsn. destruct
( isdeceqat ( natruncautoimage p q' ) ( S upper ) ) as [ i0 | i1
]. split with ( natruncautoimage p ( isreflnatleh ( S upper ) )
). split. assert ( natleh ( natruncautoimage p ( isreflnatleh ( S
upper ) ) ) ( S upper ) ) as aux. apply
natruncautoimageineq. destruct ( natlehchoice _ _ aux ) as [ l | r
]. apply natlthnsn. assumption. assert ( n ~> S upper ) as
f0. rewrite <- ( natruncautoimagepath p q' ). rewrite i0. rewrite
<- r. rewrite ( natruncautoimagepath p ( isreflnatleh ( S upper )
)). rewrite r. apply idpath. assert empty. apply ( isirreflnatlh ( S
upper ) ). apply ( natlehlthtrans _ upper ). rewrite <-
f0. assumption. apply natlthnsn. contradiction.

```

```

split. apply ( natruncautoisnj p ). apply natlthtoleh. apply (
natlehlthtrans _ upper ). apply truncnattruncautobound. destruct
( natlehchoice _ _ ( natruncautoimageineq p ( isreflnatleh ( S
upper ) ) ) ) as [ l | r ]. apply natlthnsn. assumption. assert
empty. assert ( S upper ~> n ) as f0. rewrite <-
( natruncautoimagepath p ( isreflnatleh ( S upper ) ) ). rewrite
r. rewrite <- i0. rewrite ( natruncautoimagepath p q' ). apply
idpath. apply ( isirreflnatlh ( S upper ) ). apply ( natlehlthtrans
_ upper ). rewrite f0. assumption. apply
natlthnsn. contradiction. apply natlthnsn. assumption. unfold
truncnattruncauto. destruct ( isdeceqat ( natruncautoimage p
( isreflnatleh ( S upper ) ) ) ) as [ l | r ]. assert empty. assert ( S
upper ~> n ) as f0. rewrite <- ( natruncautoimagepath p ( isreflnatleh
( S upper ) ) ). rewrite l. rewrite <- i0. rewrite (
natruncautoimagepath p q' ). apply idpath. apply
isirreflnatlh ( S upper ). apply ( natlehlthtrans _ upper
)

```

```

). rewrite f0. assumption. apply natlthnsn. contradiction. destruct
( natlthorgeh ( i ( natruncautoimage p ( isreflnatleh ( S upper
) ) ) ) ( S upper ) ) as [ l' | r' ]. assert empty. apply (
isirreflnatlh ( S upper ) ). rewrite ( natruncautoimagepath p
) in l'. assumption. contradiction. destruct ( natgehchoice _ _ r' )
as [ l' | r' ]. assert empty. apply ( isirreflnatlh ( S upper ) )
. rewrite ( natruncautoimagepath p ) in
l''. assumption. contradiction. rewrite <- i0. rewrite (
natruncautoimagepath p q' ). apply idpath. intros x y. apply
( natruncautoisnj p ). apply natruncautoimageineq. apply
natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
natlthnsn. unfold truncnattruncauto in y. destruct ( natlthorgeh (
i x ) ( S upper ) ) as [ l | r ]. assert ( S upper ~> x ) as
f0. apply ( natruncautoisnj p ). apply isreflnatleh. apply
natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
natlthnsn. rewrite <- i0. rewrite y. rewrite (
natruncautoimagepath p q' ). apply idpath. assert empty. apply
( isirreflnatlh ( S upper ) ). apply ( natlehlthtrans _ upper
). rewrite f0. assumption. apply natlthnsn. contradiction. destruct
( isdeceqat x ( S upper ) ) as [ l' | r' ]. assert empty. apply
( isirreflnatlh ( S upper ) ). apply ( natlehlthtrans _ upper
). rewrite <- l'. assumption. apply
natlthnsn. contradiction. destruct ( natgehchoice _ _ r ) as [ l' |
r' ]. assert empty. apply ( isirreflnatlh n ). apply
natlehlthtrans _ ( S upper ). assumption. rewrite <-
y. assumption. contradiction. rewrite ( natruncautoimagepath p
- ). rewrite r'. apply idpath. split with ( natruncautoimage p
q' ). split. destruct ( natlehchoice _ _ ( natruncautoimageineq
p q' ) ) as [ l | r ]. apply natlthnsn. assumption. assert
empty. apply i1. assumption. contradiction. split. unfold
truncnattruncauto. destruct ( natlthorgeh ( i (
natruncautoimage p q' ) ) ( S upper ) ) as [ l | r ]. apply
natruncautoimagepath. destruct ( natgehchoice _ _ r ) as [ l' |
r' ]. apply natruncautoimagepath. assert empty. apply
( isirreflnatlh ( S upper ) ). apply ( natlehlthtrans _ upper
). rewrite <- r'. rewrite ( natruncautoimagepath p q' )
). assumption. apply natlthnsn. contradiction.

```

```

intros x y. apply ( natruncautoisnj p ). apply ( pr1 p ). apply
natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
natlthnsn. rewrite ( natruncautoimagepath p q' ). unfold
truncnattruncauto in y. destruct ( natlthorgeh ( i x ) ( S upper ) )
as [ l | r ]. rewrite y. apply idpath. destruct ( isdeceqat x ( S
upper ) ) as [ l' | r' ]. assert empty. apply ( isirreflnatlh ( S
upper ) ). apply ( natlehlthtrans _ upper ). rewrite <-
l'. assumption. apply natlthnsn. contradiction. destruct
( natgehchoice _ _ r ). rewrite y. apply idpath. assert empty. apply
i1. apply ( natruncautoisnj p ). apply ( natruncautoimageineq
p ). apply isreflnatleh. rewrite ( natruncautoimagepath p q' )
. rewrite y. apply idpath. contradiction. apply
truncnattruncautobound. Defined.

```

```

Definition truncnattruncautoinv { upper : nat } { i : nat -> nat } { p :
isnattruncauto ( S upper ) i } : nat -> nat := natruncautoinv (
truncnattruncautoisauto p ).

```

```

Lemma precompwithnatcofaceisauto { upper : nat } { i : nat -> nat } { p :
isnattruncauto ( S upper ) i } { bound : natlth 0 ( natruncautoimage p
( isreflnatleh ( S upper ) ) ) i }. Proof. intros. set ( v :=
natruncautoimage p ( isreflnatleh ( S upper ) ) ) : bound.
isnattruncauto upper ( funcomp ( natcoface ( natruncautoimage p
( isreflnatleh ( S upper ) ) ) ) i ). change ( natruncautoimage p
( isreflnatleh ( S upper ) ) ) with v in bound. unfold isnattruncauto.
split. intros m q. unfold funcomp. assert ( natleh m ( S upper ) ) as aaa. apply
natlthtoleh. apply natlehlthtrans with ( m := upper
)

```

```

). assumption. exact ( natlthnsn upper ). set ( m' :=
nattruncautoimage p aaa ). destruct ( natlthorgeh m' v ) as [ l | r ]. (* CASE m' < v *) split with m'. split. apply natlthsntoleh.
apply ( natlthletrans _ v ). assumption. apply (
nattruncautoimageineq p _ ). split. unfold natcoface. rewrite
l. apply ( nattruncautoimagepath p aaa ). intros n j w. assert (
natcoface v n ^> m' ) as f0. apply pathsinv0. apply (
nattruncautoimagecanon p aaa ). apply
natcofaleh. assumption. assumption. rewrite <- f0. destruct (
natgthorleh v n ) as [ l' | r' ]. unfold natcoface. rewrite l'. apply
idpath. assert empty. apply ( isirreflnatlh v ). apply (
natlehltrans _ n ). assumption. apply ( istransnatlh _ ( S n )
). apply natlthnsn. unfold natcoface in f0. rewrite
natgehimplnatgtbfalse v n r' ) in f0. rewrite
f0. assumption. contradiction. (* CASE v <= m' *) set ( j :=
nattruncautoimagepath p aaa ). change ( nattruncautoimage p aaa ) with m' in j. set ( m' := minus m' 1 ). assert ( natleh m' upper
) as a0. destruct ( natlthorgeh 0 m' ) as [ h | h' ]. rewrite <- ( minusm1 upper ). apply minusleh. assumption. apply ( natlehltrans
upper ). apply natlehOn. apply natlthnsn. apply (
nattruncautoimageineq ). destruct ( natgechoice 0 m' h' ) as [ k | k' ]. assert empty. apply ( negnatghOn m' k ). contradiction. unfold
m''. rewrite <- k'. apply natlehOn. destruct ( natgechoice m' v r )
as [ l' | r' ]. assert ( natleh v m' ) as a2. apply
natlthsntoleh. unfold m''. rewrite pathssminus. rewrite
minusm1. assumption. destruct ( natlehchoice 0 m' ( natlehOn m' ) )
as [ k | k' ]. assumption. assert empty. apply ( negnatghOn v
). rewrite k'. assumption. contradiction. assert ( i ( natcoface v
m'' ) ^> m ) as a1. unfold natcoface. rewrite ( natgehimplnatgtbfalse
v m'' a2 ). unfold m''. rewrite pathssminus. rewrite
minusm1. assumption. destruct ( natlehchoice 0 m' ( natlehOn m' ) )
as [ k | k' ]. assumption. assert empty. apply ( negnatghOn v
). rewrite k'. assumption. contradiction. split with
m''. split. assumption. split. assumption. intros n s t. assert (
natcoface v n ^> natcoface v m'' ) as g. assert ( natcoface v n ^> m'
) as g0. apply pathsinv0. apply ( nattruncautoimagecanon p aaa
). apply natcofaleh. assumption. assumption. assert ( natcoface v
m'' ^> m' ) as g1. unfold m''. unfold nattruncautoimage. apply
pathsinv0. apply ( nattruncautoimagecanon p aaa ). apply
natcofaleh. assumption. assumption. rewrite g0, g1. apply idpath.
change ( idfun _ m'' ^> idfun _ n ). rewrite <- (
natcofacetractractractract v ). unfold funcomp. rewrite g. apply
idpath. assert empty. apply ( isirreflnatlh ( S upper ) ). apply (
natlehltrans _ upper ). assert ( S upper ^> m ) as g. rewrite <- (
nattruncautoimagepath p ( isreflnatleh ( S upper ) ) ). change ( i
v ^> m ). rewrite <- j. rewrite r'. apply idpath. rewrite
g. assumption. apply natlthnsn. contradiction.

intros x X. unfold funcomp. assert ( natleh ( i ( natcoface v x
) ( S upper ) ) as a0. apply p. apply
natcofaleh. assumption. destruct ( natlehchoice _ _ a0 ) as
[ l | r ]. apply natlthsntoleh. assumption. assert ( v ^>
natcoface v x ) as g. unfold v. apply (
nattruncautoimagecanon p ( isreflnatleh ( S upper ) )
). unfold natcoface. destruct ( natgthorleh v x ) as [ a | b
]. unfold v in a. rewrite a. apply natltholeh. apply (
natlehltrans _ upper ). assumption. apply natlthnsn. unfold v
in b. rewrite ( natgehimplnatgtbfalse _ x b ). assumption.
assumption. assert empty. destruct ( natgthorleh v x ) as [ a | b ]. unfold natcoface in g. rewrite a in g. apply (
isirreflnatlh x ). rewrite g in a. assumption. unfold natcoface
in g. rewrite ( natgehimplnatgtbfalse v x b ) in g. apply (
isirreflnatlh x ). apply ( natlthletrans _ ( S x ) ). apply
natlthnsn. rewrite <- g. assumption. contradiction. Defined.

Lemma nattruncautoimagecanon { R : commrng } { upper : nat } { i j :

```

```

nat -> nat ) ( p : isnattruncauto upper i ) ( p' : isnattruncauto
upper j ) : isnattruncauto upper ( funcomp j i ). Proof.
intros. split. intros n n'. split with ( nattruncautoimage p' (
nattruncautoimageineq p n' ) ). split. apply (
nattruncautoimageineq p' ). split. unfold funcomp. rewrite (
nattruncautoimagepath p' _ ). rewrite ( nattruncautoimagepath p
_ ). apply idpath. intros x X y y. unfold funcomp in y. apply (
nattruncautoimageineq p' ). apply
nattruncautoimageineq. assumption. apply ( nattruncautoimageineq p
). apply p'. apply nattruncautoimageineq. apply
p'. assumption. rewrite ( nattruncautoimagepath p' ). rewrite (
nattruncautoimagepath p ). rewrite y. apply idpath. intros x X.
unfold funcomp. apply p. apply p'. assumption. Defined.

```

```

Definition nattruncreverse ( upper : nat ) : nat -> nat. Proof.
intros upper n. destruct ( natgthorleh n upper ) as [ h | k ]. exact
n. exact ( minus upper n ). Defined.

```

```

Definition nattruncbottomtopswap ( upper : nat ) : nat -> nat. Proof.
intros upper n. destruct ( isdeceqnat 0%nat n ) as [ h | k ]. exact (
upper ). destruct ( isdeceqnat upper n ) as [ l | r ]. exact ( 0%nat
). exact n. Defined.

```

```

Lemma nattruncreverseisnattruncauto ( upper : nat ) : isnattruncauto
upper ( nattruncreverse upper ). Proof. intros. unfold
isnattruncauto. split. intros m q. set ( m' := minus upper m
). assert ( natleh m' upper ) as a0. apply minusleh. assert (
nattruncreverse upper m' ^> m ) as a1. unfold
nattruncreverse. destruct ( natgthorleh m' upper ). assert
empty. apply isirreflnatlh with ( n := m' ). apply natlehltrans
with ( m := upper ). assumption. assumption. contradiction. unfold
m'. rewrite doubleminuslehpaths. apply idpath. assumption. split with
m'. split. assumption. split. assumption. intros n qq u. unfold
m'. rewrite <- u. unfold nattruncreverse. destruct ( natgthorleh n
upper ) as [ l | r ]. assert empty. apply ( isirreflnatlh n ). apply
( natlehltrans _ upper ). assumption. assumption. contradiction. rewrite
doubleminuslehpaths. apply idpath. assumption. intros x X. unfold
nattruncreverse. destruct ( natgthorleh x upper ) as [ l | r
]. assumption. apply minusleh. Defined.

```

```

Lemma nattruncbottomtopswapselfinv ( upper n : nat ) :
nattruncbottomtopswap upper ( nattruncbottomtopswap upper n ) ^> n.
Proof. intros. unfold nattruncbottomtopswap. destruct ( isdeceqnat
upper n ). destruct ( isdeceqnat 0%nat n ). destruct ( isdeceqnat
0%nat upper ). rewrite <- i0. rewrite <- i1. apply idpath. assert
empty. apply e. rewrite i0. rewrite i. apply idpath. contradiction.
destruct ( isdeceqnat 0%nat 0%nat ). assumption. assert empty. apply
e0. auto. contradiction. destruct ( isdeceqnat 0%nat n ). destruct (
isdeceqnat 0%nat upper ). rewrite <- i. rewrite i0. apply idpath.
destruct ( isdeceqnat upper upper ). assumption. assert empty. apply
e1. auto. contradiction. destruct ( isdeceqnat 0%nat n ). assert
empty. apply e0. assumption. contradiction. destruct ( isdeceqnat
upper n ). assert empty. apply e. assumption. contradiction. auto.
Defined.

```

```

Lemma nattruncbottomtopswapbound ( upper n : nat ) ( p : natleh n
upper ) : natleh ( nattruncbottomtopswap upper n ) upper. Proof.
intros. unfold nattruncbottomtopswap. destruct ( isdeceqnat 0%nat n
). auto. destruct ( isdeceqnat upper n ). apply isreflnatleh. apply
isreflnatleh. destruct ( isdeceqnat upper n ). apply
natlehOn. assumption. Defined.

```

```

Lemma nattruncbottomtopswapisnattruncauto ( upper : nat ) :
isnattruncauto upper ( nattruncbottomtopswap upper ). Proof.
intros. unfold isnattruncauto. split. intros m p. set ( m' :=

```

```

natruncbottomtopswap upper m). assert ( natleh m' upper ) as
a0. apply natruncbottomtopswapbound. assumption. assert
(natruncbottomtopswap upper m' > m) as a1. apply
natruncbottomtopswapselinv. split with
m'. split. assumption. split. assumption. intros k q u. unfold
m'. rewrite <- u. rewrite natruncbottomtopswapselinv. apply
idpath. intros n p. apply natruncbottomtopswapbound. assumption.
Defined.

Lemma isnatruncautoOS { upper : nat } { i : nat -> nat } ( p :
isnatruncauto ( S upper ) i ) ( j : i 0%nat > S upper ) :
isnatruncauto upper ( funcomp S i ). Proof. intros. unfold
isnatruncauto. split. intros m q. set ( v := natruncautopreimage p
(natlehtoleh m ( S upper )) (natlehtohtrans m upper ( S upper )) q
(natlthnsn upper))). destruct ( isdeceqnat 0%nat v ) as [ i0 | i1 ]. assert empty. apply ( isirreflnat ( i 0%nat ) ). apply
natlehtohtrans _ upper. rewrite i0. unfold v. rewrite (
natruncautoimagepath ). assumption. rewrite j. apply
natlthnsn. contradiction. assert ( natlehtoh v ) as aux. destruct (
natlehchoice _ _ ( natlehtoh v ) ). assumption. assert empty. apply
i1. assumption. contradiction. split with ( minus v 1
). split. rewrite <- ( minussn1 upper ). apply ( minusileh aux
(natlehtohtrans _ _ ( natlehtoh upper )) ( natlthnsn upper ) ) (
natruncautoimageineq p ( natltholeh m ( S upper ) )
natlehtohtrans m upper ( S upper ) q ( natlthnsn upper ) ) )
). split. unfold funcomp. rewrite pathssminus. rewrite
minussn1. apply natruncautoimagepath. assumption. intros n uu
k. unfold funcomp in k. rewrite <- ( minussn1 n ). assert ( v > S n
) as f. apply ( natruncautoimagecanon p _ )
). assumption. assumption. rewrite f. apply idpath. intros x
X. unfold funcomp. assert ( natleh ( i ( S x ) ) ( S upper ) ) as
aux. apply p. assumption. destruct ( natlehchoice _ _ aux ) as [ h | k ]. apply natlthnsntoleh. assumption. assert empty. assert ( 0%nat > S x ) as ii. apply ( natruncautoinj p ). apply natlehtoh. assumption.
rewrite j. rewrite k. apply idpath. apply ( isirreflnatlh ( S x ) ). apply
(natlehtohtrans _ x ). rewrite <- ii. apply natlehtoh. apply
natlthnsn. contradiction. Defined.

(* The following lemma says that we may reindex sums along
automorphisms of the interval over which the finite summation is being
taken. *)

```

Lemma natsummationreindexing { R : commrng } { upper : nat } { i : nat -> nat } { p : isnatruncauto upper } (f : nat -> R) :
natsummation0 upper f > natsummation0 upper (funcomp i f). Proof.
intros R upper. induction upper. intros. simpl. unfold funcomp.
assert (0%nat > i 0%nat) as f0. destruct (natlehchoice (i 0%nat) 0%nat (pr2 p 0%nat (isreflnatlh 0%nat))) as [h | k]. assert
empty. exact (negnatlthm0 (i 0%nat) h). contradiction. rewrite
k. apply idpath. rewrite <- f0. apply idpath. intros. simpl (
natsummation0 (S upper) f). set (j := natruncautoimagepath p
(isreflnatleh (S upper))). set (v := natruncautoimage p
(isreflnatleh (S upper))). change (natruncautoimage p
(isreflnatleh (S upper))) with v in j. destruct (natlehchoice
0%nat v (natlehtoh v)). set (aaa := natruncautoimageineq p
(isreflnatleh (S upper))). change (natruncautoimage p
(isreflnatleh (S upper))) with v in aaa. destruct (natlehchoice
(S upper) aaa) as [l | r]. rewrite (IHupper (funcomp
(natcoface v) i)).

change (funcomp (funcomp (natcoface v) i) f) with (funcomp (
natcoface v) (funcomp i f)). assert (f (S upper) >
(funcomp i f) v) as f0. unfold funcomp. rewrite j. apply
idpath. rewrite f0.

assert (natleh v upper) as aux. apply natlthnsntoleh. assumption.

```

rewrite ( natsummationshift upper ( funcomp i f ) aux ). apply
idpath. apply precompwithnatcofaceisauto. assumption.

rewrite ( IHupper ( funcomp ( natcoface v ) i ) ). assert (
natsummation0 upper ( funcomp ( funcomp ( natcoface v ) i ) f ) >
natsummation0 upper ( funcomp i f ) ) as f0. apply
natsummationpathsuperfixed. intros x X. unfold funcomp. unfold
natcoface. assert ( natlh x v ) as a0. apply ( natlehtohtrans _ upper ). assumption. rewrite r. apply natlthnsn. rewrite a0. apply
idpath. rewrite f0. assert ( f ( S upper ) > ( funcomp i f ) ( S upper ) ) as f1. unfold funcomp. rewrite <- r. rewrite j. rewrite
<- r. apply idpath. rewrite f1. apply idpath. apply
precompwithnatcofaceisauto. assumption. rewrite
natsummationshift0. unfold funcomp at 2. rewrite i0. rewrite j.
assert ( i 0%nat > S upper ) as j'. rewrite i0. rewrite j. apply
idpath. rewrite ( IHupper ( funcomp S i ) ( isnatruncautoOS p j' ) )
). apply idpath. Defined.

(** * III. Formal Power Series *)

Definition seqson ( A : UU ) := nat -> A.

Lemma seqsonisaset ( A : hSet ) : isaset ( seqson A ). Proof.
intros. unfold seqson. change ( isofhlevel 2 ( nat -> A ) ). apply
impredfun. apply A. Defined.

Definition isasetfps ( R : commrng ) : isaset ( seqson R ) :=
seqsonisaset R.

Definition fps ( R : commrng ) : hSet := hSetpair _ ( isasetfps R ).

Definition fpplus ( R : commrng ) : binop ( fps R ) := fun v w n =>
( v n ) + ( w n ).

Definition fpstimes ( R : commrng ) : binop ( fps R ) := fun s t n =>
natsummation0 n ( fun x : nat => ( s x ) * ( t ( minus n x ) ) ).

(* SOME TESTS OF THE SUMMATION AND FPSTIMES DEFINITIONS: */

Definition test0 : seqson hz. Proof. intro n. induction n. exact
0. exact ( nattohz ( S n ) ). Defined.

Eval lazy in ( hzabsval ( natsummation0 1 test0 ) ).


Definition test1 : seqson hz. Proof. intro n. induction n. exact ( 1
+ 1 ). exact ( ( 1 + 1 ) * IHn ). Defined.



Eval lazy in ( hzabsval ( fpstimes hz test0 test1 0%nat ) ).



Eval lazy in ( hzabsval ( fpstimes hz test0 test1 1%nat ) ).



Eval lazy in ( hzabsval ( fpstimes hz test0 test1 2%nat ) ).



Eval lazy in ( hzabsval ( fpstimes hz test0 test1 3%nat ) ).



Eval lazy in ( hzabsval ( fpstimes hz test0 test1 4%nat ) ). */

Definition fpzero ( R : commrng ) : fps R := ( fun n : nat => 0 ).



Definition fpstone ( R : commrng ) : fps R. Proof. intros. intro
n. destruct n. exact 1. exact 0. Defined.



Definition fpssminus ( R : commrng ) : fps R -> fps R := ( fun s n => -
( s n ) ).



Lemma ismonoidopfpsplus ( R : commrng ) : ismonoidop ( fpplus R ).


```

Proof. intros. unfold ismonoidop. split. unfold isassoc. intros s t u. unfold fpsplus. (* This is a hack which should work immediately without such a workaround! *) change ((fun n : nat => s n + t n + u n) ~> (fun n : nat => s n + (t n + u n))). apply funextfun. intro n. apply R.

unfold isunital. assert (isunit (fpsplus R) (fpszero R)) as a. unfold isunit. split. unfold islunit. intro s. unfold fpsplus. unfold fpszero. change ((fun n : nat => 0 + s n) ~> s). apply funextfun. intro n. apply rngrunax1.

unfold isrunit. intro s. unfold fpsplus. unfold fpszero. change ((fun n : nat => s n + 0) ~> s). apply funextfun. intro n. apply rngrunax1. exact (tpair (fpszero R) a). Defined.

Lemma isgropfpsplus (R : commrng) : isgrop (fpsplus R). Proof. intros. unfold isgrop. assert (invstruct (fpsplus R) (ismonoidopfpsplus R)) as a. unfold invstruct. assert (isinv (fpsplus R) (unel_is (ismonoidopfpsplus R)) (fpsminus R)) as b. unfold isinv. split. unfold islinv. intro s. unfold fpsplus. unfold fpsminus. unfold unel_is. simpl. unfold fpszero. apply funextfun. intro n. exact (rngrlinax1 R (s n)). unfold isrinv. intro s. unfold fpsplus. unfold fpsminus. unfold unel_is. simpl. unfold fpszero. apply funextfun. intro n. exact (rngrlinax1 R (s n)). unfold iscommfpsplus (R : commrng) : iscomm (fpsplus R). Proof. intros. unfold iscomm. intros s t. unfold fpsplus. change ((fun n : nat => s n + t n) ~> (fun n : nat => t n + s n)). apply funextfun. intro n. apply R. Defined.

Lemma isassocfpstimes (R : commrng) : isassoc (fpstimes R). Proof. intros. unfold isassoc. intros s t u. unfold fpstimes.

assert ((fun n : nat => natsummation0 n (fun x : nat => natsummation0 (minus n x) (fun x0 : nat => s x * (t x0 * u (minus (minus n x) x0)))) ~> (fun n : nat => natsummation0 n (fun x : nat => s x * natsummation0 (minus n x) (fun x0 : nat => t x0 * u (minus (minus n x) x0))))) as A. apply funextfun. intro n. apply natsummationpathsupperfixed. intros. rewrite natsummationtimesdistd. apply idpath. rewrite <- A. assert ((fun n : nat => natsummation0 n (fun x : nat => natsummation0 (minus n x) (fun x0 : nat => s x * t x0 * u (minus (minus n x) x0)))) ~> (fun n : nat => natsummation0 n (fun x : nat => natsummation0 (minus n x) (fun x0 : nat => s x * (t x0 * u (minus (minus n x) x0))))) as B. apply funextfun. intro n. apply maponpaths. apply funextfun. intro m. apply maponpaths. apply funextfun. intro x. apply R. assert ((fun n : nat => natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat => s x0 * t (minus x x0) * u (minus (minus n x) x0)))) ~> (fun n : nat => natsummation0 n (fun x : nat => natsummation0 (minus n x) (fun x0 : nat => s x * t x0 * u (minus (minus n x) x0))))) as C. apply funextfun. intro n. set (f := fun x : nat => (fun x0 : nat => s x * t x0 * u (minus (minus n x) x0))). assert (natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat => f x0 (minus x x0))) ~> (natsummation0 n (fun x : nat => natsummation0 (minus n x) (fun x0 : nat => f x0 x0)))) as D. apply natsummatonswap. unfold f in D. assert (natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat => s x0 * t (minus x x0) * u (minus n x))) ~> natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat => s x0 * t (minus x x0) * u (minus (minus n x0) (minus x x0))))) as E. apply natsummationpathsupperfixed. intros k p. apply natsummationpathsupperfixed. intros l q. rewrite (natdoubleminus p q). apply idpath. rewrite E, D. apply idpath. rewrite <- B. rewrite <- C. assert ((fun n : nat => natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat => s x0 * t (minus x x0) * u (minus n x))) ~> (fun n : nat => natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat => s x0 * t (minus x x0) * u (minus n x))))) as D. apply funextfun. intro n. apply maponpaths. apply funextfun. intro m. apply natsummationtimesdistl. rewrite <- D. apply idpath. Defined.

Lemma natsummationandfpszero (R : commrng) : forall m : nat, natsummation0 m (fun x : nat => fpszero R x) ~> (@rngrunell R). Proof. intros R m. induction m. apply idpath. simpl. rewrite IHm. rewrite (rngrunax1 R). apply idpath. Defined.

Lemma ismonoidopfpstimes (R : commrng) : ismonoidop (fpstimes R). Proof. intros. unfold ismonoidop. split. apply isassocfpstimes. split with (fpsone R). split. intro s. unfold fpstimes. change ((fun n : nat => natsummation0 n (fun x : nat => fpsone R x * s (minus n x))) ~> s). apply funextfun. intro n. destruct n. simpl. rewrite (rngrlinax2 R). apply idpath. rewrite natsummationshift0. rewrite (rngrlinax2 R). rewrite minusOr. assert (natsummation0 n (fun x : nat => fpsone R (S x) * s (minus n x)) ~> ((natsummation0 n (fun x : nat => fpszero R)))) as f. apply natsummationpathsupperfixed. intros m m'. rewrite (rngrmult0x R). apply idpath. change (natsummation0 n (fun x : nat => fpsone R (S x) * s (minus n x)) + s (S n) ~> (s (S n))). rewrite f. rewrite natsummationandfpszero. apply (rngrlinax1 R). intros s. unfold fpstimes. change ((fun n : nat => natsummation0 n (fun x : nat => s x * fpsone R (minus n x))) ~> s). apply funextfun. intro n. destruct n. simpl. rewrite (rngrlinax2 R). apply idpath. change (natsummation0 n (fun x : nat => s x * fpsone R (minus n n)) ~> s (S n)). rewrite minusSn. rewrite (rngrlinax2 R). assert (natsummation0 n (fun x : nat => s x * fpsone R (minus (S n) x)) ~> ((natsummation0 n (fun x : nat => fpsone R x)))) as f. apply natsummationpathsupperfixed. intros m m'. rewrite <- pathssminus. rewrite (rngrmult0x R). apply idpath. apply (natlehlthtrans _ n). assumption. apply natithmsn. rewrite f. rewrite natsummationandfpszero. apply (rngrlinax1 R). Defined.

Lemma iscommfpstimes (R : commrng) (s t : fps R) : fpstimes R s t ~> fpstimes R t s. Proof. intros. unfold fpstimes. change ((fun n : nat => natsummation0 n (fun x : nat => s x * t (minus n x))) ~> (fun n : nat => natsummation0 n (fun x : nat => t x * s (minus n x))). apply funextfun. intro n.

assert (natsummation0 n (fun x : nat => s x * t (minus n x)) ~> (natsummation0 n (fun x : nat => t (minus n x) * s x))) as a0. apply maponpaths. apply funextfun. intro m. apply R. assert ((natsummation0 n (fun x : nat => t (minus n x) * s x)) ~> (natsummation0 n (funcomp (nattruncreverse n) (fun x : nat => t x * s (minus n x))))) as a1.

apply natsummationpathsupperfixed. intros m q. unfold funcomp. unfold nattruncreverse. destruct (natgthorleh m n). assert empty. apply isirreflatlh with (n := n). apply natlthletrans with (m := m). apply h. assumption. contradiction. apply maponpaths. apply maponpaths. apply pathsinv0. apply doubleminuslepaths. assumption. assert ((natsummation0 n (funcomp (nattruncreverse n) (fun x : nat => t x * s (minus n x))) ~> natsummation0 n (fun x : nat => t x * s (minus n x))) as a2. apply pathsinv0. apply natsummationreindexing. apply

```

natattruncreverseisnatattrunauto. exact ( pathscomp0 a0 ( pathscomp0 a1
a2 ) ). Defined.

Lemma isldistrfps ( R : commrng ) ( s t u : fps R ) : fpstimes R s (
fpsplus R t u ) ~> ( fpsplus R ( fpstimes R s t ) ( fpstimes R s u ) )
. Proof. intros. unfold fpstimes. unfold fpstimes. change ((fun n :
nat => natsummation0 n (fun x : nat => s x * (t (minus n x) + u (
minus n x)))) ~> (fun n : nat => natsummation0 n (fun x : nat => s x *
t (minus n x)) + natsummation0 n (fun x : nat => s x * u (minus n x)))
). apply funextfun. intro upper. assert ( natsummation0 upper (fun x :
nat => s x * (t (minus upper x) + u (minus upper x))) ) ~> ( natsummation0 upper (fun x : nat => ((s x * t (minus upper x)) +
(s x * u (minus upper x))))) as a0. apply maponpaths. apply
funextfun. intro n. apply R. assert (( natsummation0 upper (fun x :
nat => ((s x * t (minus upper x)) + (s x * u (minus upper x))))) ~> (( natsummation0 upper (fun x : nat => s x * t (minus
upper x)) ) + ( natsummation0 upper (fun x : nat => s x * u (minus
upper x))))) as a1. apply natsummationplusdistr. exact (
pathscomp0 a0 a1 ). Defined.

Lemma isrdistrfps ( R : commrng ) ( s t u : fps R ) : fpstimes R (
fpsplus R t u ) s ~> ( fpsplus R ( fpstimes R t s ) ( fpstimes R u s )
). Proof. intros. unfold fpstimes. unfold fpstimes. change ((fun n :
nat => natsummation0 n (fun x : nat => (t x + u x) * s (minus n x))) ~> (fun n : nat => natsummation0 n (fun x : nat => t x * s (minus
n x)) + natsummation0 n (fun x : nat => u x * s (minus n x))). apply funextfun. intro upper. assert ( natsummation0 upper (fun x :
nat => (t x + u x) * s (minus upper x)) ~> ( natsummation0 upper
( fun x : nat => ((t x * s (minus upper x)) + (u x * s (minus
upper x))))) ) as a0. apply maponpaths. apply funextfun. intro n. apply R. assert (( natsummation0 upper (fun x : nat => ((t x *
s (minus upper x)) + (u x * s (minus upper x))))) ~> (( natsummation0 upper (fun x : nat => t x * s (minus upper x)) ) + ( natsummation0 upper (fun x : nat => u x * s (minus upper x))))) as a1. apply natsummationplusdistr. exact ( pathscomp0 a0 a1 ). Defined.

Definition fpssrng ( R : commrng ) := setwith2binoppair ( hSetpair (
seqson R ) ( isasetfps R ) ) ( dirprodpair ( fpsplus R ) ( fpstimes R ) ).

Theorem fpssiscommrng ( R : commrng ) : iscommrng ( fpssrng R ). Proof.
intro. unfold iscommrng. unfold iscommrngops. split. unfold
isrngops. split. split. unfold isabrop. split. exact ( isgropfpplus
R ). exact ( iscommfpplus R ). exact ( ismonoidopfpstimes R ). unfold
isidistr. split. unfold isldistr. intros. apply ( isldistrfps R ). unfold
isrdistr. intros. apply ( isrdistrfps R ). unfold
iscomm. intros. apply ( iscommfpstimes R ). Defined.

Definition fpsscommrng ( R : commrng ) : commrng := commrngpair (
fpssrng R ) ( fpssiscommrng R ).

Definition fpsshift ( R : commrng ) ( a : fpsscommrng R ) : fpsscommrng
R := fun n : nat => a ( S n ).

Lemma fpsshiftandmult { R : commrng } ( a b : fpsscommrng R ) ( p : b
0?nat > 0 ) : forall n : nat, ( a * b ) ( S n ) ~> (( a * ( fpsshift
b ) ) n ). Proof. intros. induction n. change ( a * b ) with (
fpstimes R a b ). change ( a * fpsshift b ) with ( fpstimes R a (
fpsshift b ) ). unfold fpstimes. unfold fpsshift. simpl. rewrite
p. rewrite ( rngmultx0 R ). rewrite ( rngrunax1 R ). apply idpath.
change ( a * b ) with ( fpstimes R a b ). change ( a * fpsshift b )
with ( fpstimes R a ( fpsshift b ) ). unfold fpsshift. unfold
fpstimes. change ( natsummation0 ( S ( S n )) (fun x : nat => a x * b
(minus ( S ( S n )) x)) ) with ( ( natsummation0 ( S n ) (fun x : nat
=> a x * b (minus ( S ( S n )) x)) ) + a ( S ( S n )) * b (minus
( S ( S n )) ( S ( S n )) ) ). rewrite minusnn0. rewrite p. rewrite
( rngmultx0 R ). rewrite rngrunax1. apply
natsummationpathsupperfixed. intros x j. apply maponpaths. apply
maponpaths. rewrite pathssminus. apply idpath. apply ( natlehlthtrans
_ ( S n ) _ ). assumption. apply natlthnsn. Defined.

(** * IV. Apartness relation on formal power series *)

Lemma apartbinarysum0 ( R : acommrng ) ( a b : R ) ( p : a + b # 0 ) :
hdijj ( a # 0 ) ( b # 0 ). Proof. intros. intros P s. apply (
acommrng_acotrans R ( a + b ) a 0 p ). intro k. destruct k as [ l | r ]
. apply s. apply ii2. assert ( a + b # a ) as l'. apply l. assert (
( a + b ) # ( a + 0 ) ) as l''. rewrite rngrunax1. assumption. apply
( ( pr1 ( acommrng_aadd R ) ) a b 0 ). assumption. apply s. apply
ii1. assumption. Defined.

Lemma apartnatsummation0 ( R : acommrng ) ( upper : nat ) ( f : nat ->
R ) ( p : ( natsummation0 upper f ) # 0 ) : heists ( fun n : nat =>
dirprod ( natleh n upper ) ( f n # 0 ) ). Proof. intros R
upper. induction upper. simpl. intros P s. apply s. split with
0%nat. split. intros g. simpl in g. apply
nopathsfalseotent. assumption. assumption. intros. intros P s. simpl
in p. apply ( apartbinarysum0 R _ _ p ). intro k. destruct k as [ l | r ]
. apply ( IUpper f l ). intro k. destruct k as [ n ab ]. destruct
ab as [ a ]. apply s. split with n. split. apply ( iestransnatleh
_ upper _ ). assumption. apply natltholeh. apply
natlthnsn. assumption. apply s. split with ( S upper ). split. apply
isreflnatleh. assumption. Defined.

Definition fpssapart0 ( R : acommrng ) : hrel ( fpsscommrng R ) := fun s
t : fpsscommrng R => ( heists ( fun n : nat => ( s n # t n ) ) ).

Definition fpssapart ( R : acommrng ) : apart ( fpsscommrng R ). Proof.
intros. split with ( fpssapart0 R ). split. intros s f. assert (
hfalse ) as i. apply f. intro k. destruct k as [ n p ]. apply (
acommrng_airrefl R ( s n ) p ). apply i. split. intros s t p P
j. apply p. intro k. destruct k as [ n q ]. apply j. split with
n. apply ( acommrng_asym R ( s n ) ( t n ) q ). intros s t u p P
j. apply p. intro k. destruct k as [ n q ]. apply ( acommrng_acotrans
R ( s n ) ( t n ) ( u n ) q ). intro l. destruct l as [ l | r ]. apply
j. apply iii. intros v V. apply V. split with n. assumption.
apply j. apply ii2. intros v V. apply V. split with n. assumption.
Defined.

Lemma fpssapartisbinopapartplus1 ( R : acommrng ) : isbinopapartl (
fpssapart R ) ( @op1 ( fpsscommrng R ) ). Proof. intros. intros a b c
p. intros P s. apply p. intro k. destruct k as [ n q ]. apply
s. change ( ( a + b ) n ) with ( ( a n ) + ( b n ) ) in q. change ( ( a
+ c ) n ) with ( ( a n ) + ( c n ) ) in q. split with n. apply ((
pri1 ( acommrng_aadd R ) ) ( a n ) ( b n ) ( c n ) q ). Defined.

Lemma fpssapartisbinopapartplus2 ( R : acommrng ) : isbinopapartc (
fpssapart R ) ( @op2 ( fpsscommrng R ) ). Proof. intros. intros a b c
p. rewrite ( rngcomm1 ( fpsscommrng R ) ) in p. rewrite ( rngcomm1 (
fpsscommrng R ) c ) in p. apply ( fpssapartisbinopapartplus1 _ a b c
). assumption. Defined.

Lemma fpssapartisbinopapartmult1 ( R : acommrng ) : isbinopapartl (
fpssapart R ) ( @op2 ( fpssrng R ) ). Proof. intros. intros a b c
p. intros P s. apply p. intro k. destruct k as [ n q ]. change ( ( a
* b ) n ) with ( natsummation0 n (fun x : nat => ( a x ) * ( b (
minus n x ) ) ) ) in q. change ( ( a * c ) n ) with ( natsummation0 n
( fun x : nat => ( a x ) * ( c ( minus n x ) ) ) ) in q. assert (
natsummation0 n (fun x : nat => ( a x * b (minus n x) - ( a x * c (
minus n x ) ) ) # 0 ) as q'. assert ( natsummation0 n (fun x : nat
=> ( a x * b (minus n x ) ) ) - natsummation0 n (fun x : nat => ( a

```

```

x * c ( minus n x ) ) # 0 ) as q''. apply aaminuszero. assumption.
assert ( (fun x : nat => a x * b (minus n x) - a x * c ( minus n x))
~> (fun x : nat => a x * b ( minus n x) + (- 1%rng) * ( a x * c ( minus n x) ) ) as i. apply funextfun. intro x. apply
maponpaths. rewrite <- ( rngmultwithminus1 R ). apply idpath. rewrite
i. rewrite natsummationplusdistr. rewrite <- ( natsummationtimesdistr
n ( fun x : nat => a x * c ( minus n x) ) ( - 1%rng ) ). rewrite ( rngmultwithminus1 R ). assumption. apply ( apartnatsummation0 R n _ q' ). intro k. destruct k as [ m g ]. destruct g as [ g g' ]. apply
s. split with ( minus n m ). apply ( ( pr1 ( acommrngaumult R ) ) ( a m ) ( b ( minus n m ) ) ( c ( minus n m ) ) ). apply
aminuszeroa. assumption. Defined.

```

```

Lemma fpssapartisbinopapartmult ( R : acommrnga ) : isbinopapart ( fpssapart R ) ( @op2 ( fpssrung R ) ). Proof. intros. intros a b c p. rewrite ( rngcomm2 ( fpsscommrng R ) ) in p. rewrite ( rngcomm2 ( fpsscommrng R ) c ) in p. apply ( fpssapartisbinopapartmultl _ a b c ). assumption. Defined.

```

```

Definition acommrngafps ( R : acommrnga ) : acommrnga. Proof.
intros. split with ( fpsscommrng R ). split with ( fpssapart R ). split. split. apply ( fpssapartisbinopapartplusl R ). apply ( fpssapartisbinopapartplusr R ). split. apply ( fpssapartisbinopapartmultl R ). apply ( fpssapartisbinopapartmultr R ). Defined.

```

```

Definition isacommrnga partdec ( R : acommrnga ) := isapartdec ( ( pr1 ( pr2 R ) ) ).
```

```

Lemma leadingcoefficientpartdec ( R : aintdom ) ( a : fpsscommrng R )
(is : isacommrnga partdec R ) ( p : a 0%nat # 0 ) : forall n : nat,
forall b : fpsscommrng R, ( b n # 0 ) -> ( ( acommrnga partrel ( acommrngafps R ) ) ( a * b ) 0 ). Proof. intros R a is p n. induction
n. intros b q. intros P s. apply s. split with 0%nat. change ( ( a *
b ) 0%nat ) with ( ( a 0%nat ) * ( b 0%nat ) ). apply

```

```

R. assumption. assumption. intros b q. destruct ( is ( b 0%nat ) 0 )
as [ left | right ]. intros P s. apply s. split with 0%nat. change ( ( a * b ) 0%nat ) with ( ( a 0%nat ) * ( b 0%nat ) ). apply
R. assumption. assumption. assert ( ( acommrnga partrel ( acommrngafps R ) ) ( a * ( fpsshift b ) ) 0 ) as j. apply IHn. unfold
fpsshift. assumption. apply j. intro k. destruct k as [ k i ]. intros
P s. apply s. rewrite <- ( fpsshiftandmult a b right k ) in i. split
with ( S k ). assumption. Defined.

```

```

Lemma apartdecintdom0 ( R : aintdom ) ( is : isacommrnga partdec R ) :
forall n : nat, forall a b : fpsscommrng R, ( a n # 0 ) -> ( acommrnga partrel ( acommrngafps R ) b 0 ) -> ( acommrnga partrel ( acommrngafps R ) ( a * b ) 0 ). Proof. intros R is n. induction
n. intros a b p q. apply q. intros k. destruct k as [ k k0 ]. apply
leadingcoefficientpartdec R a is p k . assumption. intros a b p q. destruct ( is ( a 0%nat ) 0 ) as [ left | right ]. apply q. intros
k. destruct k as [ k k0 ]. apply ( leadingcoefficientpartdec R a is
left k ). assumption. assert ( acommrnga partrel ( acommrngafps R ) ( ( fpsshift a ) * b ) 0 ) as i. apply IHn. unfold
fpsshift. assumption. assumption. apply i. intros k. destruct k as [ k k0 ]. intros P s. apply s. split with ( S k ). rewrite
rngcomm2. rewrite fpsshiftandmult. rewrite
rngcomm2. assumption. assumption. Defined.

```

```

Theorem apartdecoisaintdomfps ( R : aintdom ) ( is :
isacommrnga partdec R ) : aintdom. Proof. intros R. split with ( acommrngafps R ). split. intros P s. apply s. split with 0%nat. change
( ( @rngunel1 ( fpsscommrng R ) ) 0%nat ) with ( @rngunel1 R ). change
( @rngunel2 R # ( @rngunel1 R ) ). apply R. intros a b p q. apply
p. intro n. destruct n as [ n n0 ]. apply ( apartdecintdom0 R is n )
. assumption. assumption. Defined.

```

Close Scope rng_scope.

(** END OF FILE **)

7.5 The file frac.v

```

(** *The Heyting field of fractions for an apartness domain *)
(** By Alvaro Pelayo, Vladimir Voevodsky and Michael A. Warren *)
(** February 2011 and August 2012 *)
(** Settings *)
Add Rec LoadPath "../Generalities". Add Rec LoadPath "../hlevel1".
Add Rec LoadPath "../hlevel2". Add Rec LoadPath "../Algebra".
Unset Automatic Introduction. (** This line has to be removed for the
file to compile with Coq8.2 *)
(** Imports *)
Require Export lemmas.
(** * I. The field of fractions for an integrable domain with an
apartness relation *)
Open Scope rng_scope.

```

Section aint.

Variable A : aintdom.

```

Ltac permute := solve [ repeat rewrite rngassoc2; match goal with | [
|- ?X ~> ?X ] => apply idpath | [ |- ?X * ?Y ~> ?X * ?Z ] => apply
maponpaths; permute | [ |- ?Y * ?X ~> ?Z * ?X ] => apply (
maponpaths ( fun x => x * X )); permute | [ |- ?X * ?Y ~> ?Y * ?X ]
=> apply rngcomm2 | [ |- ?X * ?Y ~> ?K ] => solve [ repeat rewrite
<- rngassoc2; match goal with | [ |- ?H ~> ?V * X ] => rewrite (
@rngcomm2 A V X ); repeat rewrite rngassoc2; apply maponpaths;
permute end | repeat rewrite rngassoc2; match goal with | [ |- ?H ~>
?Z * ?V ] => repeat rewrite <- rngassoc2; match goal with | [ |- ?W
* Z ~> ?L ] => rewrite ( @rngcomm2 A W Z ); repeat rewrite
rngassoc2; apply maponpaths; permute end end ] | [ |- ?X * ( ?Y * ?Z
) ~> ?K ] => rewrite ( @rngcomm2 A Y Z ); permute end | repeat
rewrite <- rngassoc2; match goal with | [ |- ?X * ?Y ~> ?Z * ?X ] =>
apply maponpaths; permute | [ |- ?Y * ?X ~> ?Z * ?X ] => apply (
maponpaths ( fun x => x * X )); permute | [ |- ?X * ?Y ~> ?Y * ?X ]
=> apply rngcomm2 end | apply idpath | idtac "The tactic permute
does not apply to the current goal!" .

```

Lemma azerorelcomp (cd : dirprod A (aintdomzerosubmonoid A)) (ef

```

: dirprod A ( aintdomazerosubmonoid A ) ( p : ( pr1 cd ) * ( pr1 (
pr2 ef ) )  $\rightarrow$  ( ( pr1 ef ) * ( pr1 ( pr2 cd ) ) ) ( q : ( pr1 cd ) #
0 ) : ( pr1 ef ) # 0. Proof. intros. change ( ( @op2 A ( pr1 cd ) ( pr1 ( pr2 ef ) ) )  $\rightarrow$  ( @op2 A ( pr1 ef ) ( pr1 ( pr2 cd ) ) ) ) in p. assert ( ( @op2 A ( pr1 cd ) ( pr1 ( pr2 ef ) ) ) # 0 ) as v. apply A. assumption. apply ( pr2 ( pr2 ef ) ). rewrite p in v. apply ( pr1 ( timesazerov ) ). Defined.

Lemma azerolmultcomp { a b c : A } ( p : a # 0 ) ( q : b # c ) : a * b # c * a. Proof. intros. apply aminuszeroa. rewrite <- rmgminusdistr. apply ( pr2 A ). assumption. apply aminuszero. assumption. Defined.

Lemma azerormultcomp { a b c : A } ( p : a # 0 ) ( q : b # c ) : b * a # c * a. Proof. intros. rewrite <- @rngcomm2 A b). rewrite (@rngcomm2 A c). apply ( azerolmultcomp p q). Defined.

Definition afldfracapartrelpre : hrel ( dirprod A ( aintdomazerosubmonoid A ) ) := fun ab cd : _  $\Rightarrow$  ( ( pr1 ab ) * ( pr1 ( pr2 cd ) ) ) # ( ( pr1 cd ) * ( pr1 ( pr2 ab ) ) ).

Lemma afldfracapartiscomprel : iscomprel ( eqrelcommrngfrac A ( aintdomazerosubmonoid A ) ) ( afldfracapartrelpre ). Proof. intros ab cd ef gh p q. unfold afldfracapartrelpre. destruct ab as [ a b ]. destruct b as [ b b' ]. destruct cd as [ c d ]. destruct d as [ d d' ]. destruct ef as [ e f ]. destruct f as [ f f' ]. destruct gh as [ g h ]. destruct h as [ h h' ]. simpl in *.

apply uahp. intro u. apply p. intro p'. apply q. intro q'. destruct p' as [ p' j ]. destruct p' as [ i p' ]. destruct q' as [ q' j ]. destruct q' as [ i' q' ]. simpl in *.

assert ( a * f * d * i * h * i' # e * b * d * i * h * i' ) as v0. assert ( a * f * d * e * b * d ) as v0. apply azerormultcomp. apply d'. assumption. assert ( a * f * d * i # e * b * d * i ) as v1. apply azerormultcomp. assumption. assumption. assert ( a * f * d * i * h # e * b * d * i * h ) as v2. apply azerormultcomp. apply h'. assumption. apply azerormultcomp. assumption. assumption. apply ( pr2 ( acommrng_amult A ) b ). apply ( pr2 ( acommrng_amult A ) f ). apply ( pr2 ( acommrng_amult A ) i ). apply ( pr2 ( acommrng_amult A ) i' ).

assert ( a * f * d * i * h * i'  $\rightarrow$  c * h * b * f * i * i' ) as l. assert ( a * f * d * i * h * i'  $\rightarrow$  a * d * i * f * h * i' ) as 10. change ( @op2 A f ) d ) i ) h ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A a d ) i ) f ) h ) i' ). permute. rewrite 10. rewrite j. change ( @op2 A c b ) i ) f ) h ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A c h ) b ) f ) i ) i' ). permute. rewrite l in v0. assert ( e * b * d * i * h * i'  $\rightarrow$  g * d * b * f * i * i' ) as k. assert ( @op2 A e b ) d ) i ) h ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A e h ) i' ) b ) d ) as ko. permute. change ( @op2 A e b ) d ) i ) h ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A g d ) b ) f ) i ) i' ). rewrite ko. assert ( @op2 A ( @op2 A ( @op2 A ( @op2 A g f ) i' )  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A g f ) i' ) ) as j''. assumption. rewrite j''. permute. rewrite k in v0. assumption.

intro u. apply p. intro p'. apply q. intro q'. destruct p' as [ p' j ]. destruct p' as [ i p' ]. destruct q' as [ q' j ]. destruct q' as [ i' q' ]. simpl in *.

assert ( c * h * b * f * i * i' # g * d * b * f * i * i' ) as v. apply azerormultcomp. apply q'. apply azerormultcomp. apply p'. apply azerormultcomp. apply f'. apply azerormultcomp. apply b'. assumption. apply ( pr2 ( acommrng_amult A ) d ). apply ( pr2 ( acommrng_amult A ) h ). apply ( pr2 ( acommrng_amult A ) i ). apply ( pr2 ( acommrng_amult A ) i' ).

assert ( c * h * b * f * i * i'  $\rightarrow$  a * f * d * h * i * i' ) as k0. assert ( @op2 A c h ) b ) f ) i ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A c b ) i ) f ) h ) i' ). permute. rewrite k0. rewrite <- j. change ( @op2 A a d ) i ) f ) h ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A a f ) d ) h ) i' ). permute. rewrite k in v. assert ( g * d * b * f * i * i'  $\rightarrow$  e * b * d * h * i * i' ) as l1. assert ( g * d * b * f * i * i'  $\rightarrow$  g * f * i * d * i * b ) as l0. change ( @op2 A g d ) b ) f ) i ) i'  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A ( @op2 A g f ) i' ) d ) b ) as l. permute. rewrite <- j'. change ( @op2 A ( @op2 A ( @op2 A ( @op2 A e h ) i' ) d ) i ) b  $\rightarrow$  @op2 A ( @op2 A ( @op2 A ( @op2 A e b ) d ) h ) i ) i' ). permute. rewrite l in v. assumption. Defined.

(** We now arrive at the apartness relation on the field of fractions itself.*)

Definition afldfracapartrel := quotrel afldfracapartiscomprel.

Lemma isirreflafldfracapartrelpre : isirrefl afldfracapartrelpre. Proof. intros ab. apply acommrng_airrefl. Defined.

Lemma issymmafldfracapartrelpre : issymm afldfracapartrelpre. Proof. intros ab cd. apply ( acommrng_asymm A ). Defined.

Lemma iscotransafldfracapartrelpre : iscotrans afldfracapartrelpre. Proof. intros ab cd ef p. destruct ab as [ a b ]. destruct b as [ b b' ]. destruct cd as [ c d ]. destruct d as [ d d' ]. destruct ef as [ e f ]. destruct f as [ f f' ]. assert ( a * f * d * e * b * d ) as v. apply azerormultcomp. assumption. assumption. apply ( acommrng_acotrans A ( a * f * d ) ( c * b * f ) ( e * b * d ) ) v ). intro u. intros P k. apply k. unfold afldfracapartrelpre in *. simpl in *. destruct u as [ left | right ]. apply iii. apply ( pr2 ( acommrng_amult A ) f ). assert ( @op2 A ( @op2 A a f ) d  $\rightarrow$  @op2 A ( @op2 A a d ) f ) as i. permute. change ( @op2 A ( @op2 A a d ) f # @op2 A ( @op2 A c b ) f ). rewrite <- i. assumption. apply ii2. apply ( pr2 ( acommrng_amult A ) b ). assert ( @op2 A ( @op2 A c f ) b  $\rightarrow$  @op2 A ( @op2 A c b ) f ) as i. permute. change ( @op2 A ( @op2 A c f ) b # @op2 A ( @op2 A e d ) f ). rewrite i. assert ( @op2 A ( @op2 A e d ) b  $\rightarrow$  @op2 A ( @op2 A e b ) d ) as j. permute. change ( @op2 A ( @op2 A c b ) f # @op2 A ( @op2 A e d ) b ). rewrite j. assumption. Defined.

Lemma isapartafldfracapartrel : isapart afldfracapartrel. Proof. intros. split. apply isirreflquotrel. exact ( isirreflafldfracapartrelpre ). split. apply issymmquotrel. exact ( issymmafldfracapartrelpre ). apply iscotransquotrel. exact ( iscotransafldfracapartrelpre ). Defined.

Definition afldfracapart : apart ( commrngfrac A ( aintdomazerosubmonoid A ) ). Proof. intros. unfold apart. split with afldfracapartrel. exact isapartafldfracapartrel. Defined.

Lemma isbinapartlafldfracop1 : isbinopapart afldfracapart op1. Proof. intros. unfold isbinopapart. assert ( forall a b c : commrngfrac A ( aintdomazerosubmonoid A ), isprop ( pri (afldfracapart) ( commrngfracop1 A ( aintdomazerosubmonoid A ) a b ) ( commrngfracop1 A ( aintdomazerosubmonoid A ) a c )  $\rightarrow$  pri (afldfracapart) b c ) as int. intros a b c. apply impred. intro p. apply ( pri ( afldfracapart ) b c ). apply ( setquotuniv3prop _ ( fun a b c => HProppair _ ( int a b c ) ) ). intros ab cd ef

```

۲۵

```

p. destruct ab as [ a b ]. destruct b as [ b' b' ]. destruct cd as [ c
d ]. destruct d as [ d' d' ]. destruct ef as [ e f ]. destruct f as [
f' f' ]. unfold afldfracpart in *. simpl. unfold
afldfracpart. unfold quotrel. rewrite setquotuniv2comm. unfold
afldfracpartrelprel. simpl.

```

```

assert ( afldfracapartrelpre ( dirprodpair ( @op1 A ( @op2 A d a ) ( @op2 A B c b ) ) ( @op ( aintdomazerousubmonoid A ) ( tpair b' b' ) ( tpair d' d' ) ) ) ( dirprodpair ( @op1 A ( @op2 A f a ) ( @op2 A b e ) ) ( @op ( aintdomazerousubmonoid A ) ( tpair b' b' ) ( tpair f' f' ) ) ) ) as u. apply p. repeat afldfracapartrelpre in u. simpl in u. rewrite 2! ( @rndistr A ) in u. repeat rewrite <- @rndassoc2 in u. assert ( (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A))) d a) b) f) ) ~> (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop => @iscommrnrngops ( @isetwith2binop X ) ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A)) f a) b) d) ) as i. permute. rewrite i in u. assert ( (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop => @iscommrnrngops ( @isetwith2binop X ) ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A)) f a) b) d) ) as j. permute. rewrite j in u. assert ( (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop => @iscommrnrngops ( @isetwith2binop X ) ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A)) c b) c) b) ) ~> (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop => @iscommrnrngops ( @isetwith2binop X ) ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A)) c f) b) b) ) as k. permute. rewrite k in u. assert ( (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng (plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop => @iscommrnrngops ( @isetwith2binop X ) ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A)) b e) b) d) ) ~> (@op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( prirng ( commrnrngtormg ( acommrnrngtocommrrng ( plaintdom A)))) ( @op2 ( @pr1 setwith2binop ( fun X : setwith2binop => @iscommrnrngops ( @isetwith2binop X ) ( @op1 X ) ( @op2 X )) ( acommrnrngtocommrrng ( plaintdom A)) e d) b) b) ) as l. permute. rewrite l in u. apply ( pr2 ( acommrnrng_amult A ) b ). apply ( pr2 ( acommrnrng_amult A ) b ). apply ( pri ( acommrnrng_aadd A ) ( f * a * b * d ) ). assumption. Defined.

```

```

Lemma isbinapartrafldfracop1 : isbinopapartr afldfracpart op1.
Proof. intros a b c. rewrite ( rngcomm1 ). rewrite ( rngcomm1 _ c ).
apply isbinapartlafldfracop1. Defined.

```

```

Lemma isbinapartlafldfracop2 : isbinapartl afldfracpart op2.
Proof. intros. unfold isbinapartl. assert ( forall a b c :
  commrgrfrac A ( aintdomazerosubmonoid A ), isaprop ( pri
  (afldfracpart ) ( commrgrfrac2 A ( aintdomazerosubmonoid A ) a b ) (
  commrgrfracop2 A ( aintdomazerosubmonoid A ) a c ) ) /> pr1
  (afldfracpart ) b c ) as int. intros a b c. apply impred. intro
  p. apply ( pri ( afldfracpart ) b c ). apply ( setquotuinv3prop _ (
  fun a b c => hPropprain _ ( int a b c ) ) ). intros ab cd ef p.

```

```
destruct ab as [ a b ]. destruct b as [ b b' ]. destruct cd as [ c  
d ]. destruct d as [ d d' ]. destruct ef as [ e f ]. destruct f as  
[ f f' ].
```

```

assert ( afldfracapartrelpre ( dirprodpair ( ( a * c ) ) ( @op (
aintdomzerosubmonoid A ) ( tpair b b' ) ( tpair d d' ) ) ) (
dirprodpair ( a * e ) ( @op ( aintdomzerosubmonoid A ) ( tpair b b' ) ( tpair f f' ) ) ) ) as u. apply p. unfold afldfracapart in

```

```
*. simpl. unfold afldfracapartrel. unfold quotrel. rewrite ( setquotuniv2comm ( eqlcommlimrnfraction A ( aindomazerosubmonoid A ) ) ). unfold afldfracapartrelpre in *. simpl. simpl in u. apply ( pr2 ( acommrng_amult A ) a ). apply ( pr2 ( acommrng_amult A ) b ).
```

```

assert ( c * f * a * b >~ (@op2 (@pr1 setwith2binop (fun X :
setwith2binop => @iscommrnrngops (priisetwith2binop X) (@op1 X) (@op2 X)) (acommrnrngtocommrgn (priaintdom A))) @op2 (@pr1 setwith2binop (fun X : setwith2binop => @iscommrnrngops (priisetwith2binop X) (@op1 X) (@op2 X)) (acommrnrngtocommrgn (priaintdom A))) a c) @op2 (@pr1 setwith2binop (fun X : setwith2binop => @iscommrnrngops (priisetwith2binop X) (@op1 X) (@op2 X)) (acommrnrngtocommrgn (priaintdom A))) b f)) ) as i. change ( c * f * a * b >~ a * c * ( b * f ) ) . permute . change ( c * f * a * b # e * d * a * b ) . rewrite i.
assert ( e * d * a * b >~ (@op2 (@pr1 setwith2binop (fun X :
setwith2binop => @iscommrnrngops (priisetwith2binop X) (@op1 X) (@op2 X)) (acommrnrngtocommrgn (priaintdom A))) @op2 (@pr1 setwith2binop (fun X : setwith2binop => @iscommrnrngops (priisetwith2binop X) (@op1 X) (@op2 X)) (acommrnrngtocommrgn (priaintdom A))) a e) @op2 (@pr1 setwith2binop (fun X : setwith2binop => @iscommrnrngops (priisetwith2binop X) (@op1 X) (@op2 X)) (acommrnrngtocommrgn (priaintdom A))) b d) ) as i'. change ( e * d * a * b >~ a * e * ( b * d ) ) . permute . rewrite i' . assumption . Defined.

```

```

Lemma isbinapartrafldfracop2 : isbinopapartr (afldfracpart ) op2
Proof. intros a b c. rewrite rngcomm2. rewrite ( rngcomm2 _ c
). apply isbinapartlafldfracop2. Defined.

```

```

Definition afldfrac0 : accomrng. Proof. intros. split with (
commrngfrac A ( aintdomazerosubmonoid A )). split with (
afldfracapart ). split. split. apply ( isbinapartlafldfracop1
). apply ( isbinapartraflfracop1 ). split. apply (
isbinapartlafldfracop2 ). apply ( isbinapartraflfracop2 ). Defined

```

```

Definition afldfracmultinvint ( ab : dirprod A ( aintdomzerosubmonoid A ) ) ( is : afldfracpartrelpre ab ( dirprodpair ( @rngunell A ) ( unel ( aintdomzerosubmonoid A ) ) ) ) : dirprod A (
aintdomzerosubmonoid A ). Proof. intros. destruct ab as [ a b ]. destruct b as [ b b' ]. split with b. simpl in is. split with a. unfold afldfracpartrelpre in is. simpl in is. change ( a # 0 ). rewrite ( @rngmult0x A ) in is. rewrite ( @rngrunax2 A ) in is. assumption. Defined.

```

```

Definition afldfracmultinv ( a : afldfrac0 ) ( is : a # 0 ) :
multinvpair afldfrac0 a. Proof. intros. assert ( forall b :
afldfrac0, isaprop ( b # 0 -> multinvpair afldfrac0 b ) ) as int.
intros. apply impred. intro p. apply ( isapropmultinvpair afldfrac0 ). assert ( forall b : afldfrac0, b # 0 -> multinvpair afldfrac0 b ) as p. apply ( setquotuinvprop _ ( fun x0 => hProppair _ ( int x0 ) ) ). intros bc q. destruct bc as [ b c ]. assert ( afldfracapartrelpre ( dirprodpair b c ) ( dirprodpair ( @rngunel1 A ) ( unel ( aintdomazerosubmonoid A ) ) ) ) as is'. apply q. split with ( setquot ( eqeqlcommgrfrac A ( aintdomazerosubmonoid A ) ) ( afldfracmultinv ( dirprodpair b c ) is' ) ).
```

```

split. change ( setquotpr ( eqrelcommrngfrac A (
aintdomzerosubmonoid A ) ) ( dirprodpair ( @op2 A ( pr1 (
afldfractmultinvint ( dirprodpair b c ) is' ) ) b ) ( @op (
aintdomzerosubmonoid A ) ( pr2 ( afldfractmultinvint ( dirprodpair b
c ) is' ) ) c ) ) ) > ( commrngfracunel2 A ( aintdomzerosubmonoid A
) ). apply iscompsetquotpr. unfold commrngfracunel2int. destruct
c as [ c' c ]. simpl. apply total2toexists. split with (
carrierpair ( fun x : pr1 A => x # 0 ) 1 ( pr1 ( pr2 A ) ) ). apply
simpl. rewrite 3! ( @rngrunax2 A ). rewrite ( @rnlunax2 A ). apply
( @rncomm2 A ).
```

```

change ( setquotpr ( eqrelcommrngfrac A ( aintdomazerosubmonoid A )
) ( dirprodpair ( @op2 A b ( pr1 ( afldfracmultinv ( dirprodpair b
c ) is' ) ) ) ( @op ( aintdomazerosubmonoid A ) c ( pr2 (
afldfracmultinv ( dirprodpair b c ) is' ) ) ) ) ) ~> (
commrnfracl2 A ( aintdomazerosubmonoid A ) ). apply
iscompsetquotpr. destruct c as [ c c' ]. simpl. apply
total2toexists. split with ( carrierpair ( fun x : pr1 A => x # 0 )
1 ( pr1 ( pr2 A ) ) ). simpl. rewrite 3! ( @rngrunax2 A ). rewrite
@rngrunax2 A . apply ( @rngcomm2 A ). apply p. assumption.
Defined.

```

```

Theorem afldfracisafld : isaafield afldfrac0. Proof. intros. split.
change ( ( afldfracapartrel ) ( @rngeunel2 ( commrnfracl2 A ( aintdomazerosubmonoid A ) ) ) ) ( @rngeunel1 ( commrnfracl2 A ( aintdomazerosubmonoid A ) ) ) ). unfold afldfracapartrel. cut ( ( @op2 A ( @rngeunel2 A ) ( @rngeunel2 A ) ) # ( @op2 A ( @rngeunel1 A ) ( @rngeunel2 A ) ) ). intro v. apply v. rewrite 2! ( @rngrunax2 A
). apply A.

```

```

intros a p. apply afldfracmultinv. assumption. Defined.
Definition afldfrac := afldpair afldfrac0 afldfracisafld.
End aint.

Close Scope rng_scope.
(** END OF FILE *)

```

7.6 The file zmodp.v

```

(** *Integers mod p *)
(** By Alvaro Pelayo, Vladimir Voevodsky and Michael A. Warren *)
(** December 2011 *)
(** Settings *)
Add Rec LoadPath "../Generalities". Add Rec LoadPath "../hlevel1".
Add Rec LoadPath "../hlevel2". Add Rec LoadPath
"../Proof_of_Extensionality". Add Rec LoadPath "../Algebra".
 $\infty$ 
Unset Automatic Introduction. (** This line has to be removed for the
file to compile with Coq8.2 *)
(** Imports *)
Require Export lemmas.

Open Scope hz_scope.

(** I. Divisibility and the division algorithm *)
Definition hzdiv0 : hz -> hz -> hz -> UU := fun n m k => ( n * k ~> m
).

Definition hzdiv : hz -> hz -> hProp := fun n m => hexists ( fun k :
hz => hzdiv0 n m k ).

Lemma hzdivisrefl : isrefl ( hzdiv ). Proof. unfold
isrefl. intro. unfold hzdiv. apply total2toexists. split with
1. apply hzmultip1. Defined.

Lemma hzdivistrans : istrans ( hzdiv ). Proof. intros a b c p
q. apply p. intro k. destruct k as [ k f ]. apply q. intro
l. destruct l as [ l g ]. intros P s. apply s. unfold hzdiv0 in f,g.
split with ( k * l ). unfold hzdiv0. rewrite <- hzmultipassoc. rewrite
f. assumption. Defined.

Lemma hzdivlinearcombleft ( a b c d : hz ) ( f : a ~> ( b + c ) ) ( x
: hzdiv d a ) ( y : hzdiv d b ) : hzdiv d c . Proof. intros a b c d f
x y P s. apply x. intro x'. apply y. intro y'. destruct x' as [ k g
]. destruct y' as [ l h ]. unfold hzdiv0 in *. apply s. split with
( k + l ). rewrite hzldistr. change ( ( d * k + d * l ) % hz ~> a
). rewrite g, h, f. apply idpath. Defined.

Lemma hzdivlinearcombright ( a b c d : hz ) ( f : a ~> ( b + c ) ) ( x
: hzdiv d b ) ( y : hzdiv d c ) : hzdiv d a . Proof. intros a b c d f
x y P s. apply x. intro x'. apply y. intro y'. destruct x' as [ k g
]. destruct y' as [ l h ]. unfold hzdiv0 in *. apply s. split with
( k + l ). rewrite hzldistr. change ( ( d * k + d * l ) % hz ~> a
). rewrite hzplusr0. rewrite hzpluscomm. assumption. Defined.

Lemma divalgorithmnonneg ( n : nat ) ( m : nat ) ( p : hz0 ( nattohz m ) ) :
total2 ( fun qr : dirprod hz hz => ( ( dirprod ( nattohz n ~> ( ( nattohz m ) * ( pr1 qr ) ) + ( pr2 qr ) ) ) ( dirprod ( hzle0 ( pr2 qr ) ) ( hzlh ( pr2 qr ) ( nattohz ( m ) ) ) ) ) ). Proof. intro. intro. induction n. intros. split with
( dirprodpair 0 0 ). split. simpl. rewrite ( rngrunax1 hz ). rewrite
( rngrmult0 hz ). rewrite nattohzand0. change ( 0 ~> 0%hz ). apply
idpath. split. apply isreflhzieh. assumption.

Lemma divalgorithm ( n : nat ) ( m : nat ) ( p : hz0 ( nattohz m ) ) :
total2 ( fun qr : dirprod hz hz => ( ( dirprod ( nattohz n ~> ( ( nattohz m ) * ( pr1 qr ) ) + ( pr2 qr ) ) ) ( dirprod ( hzle0 ( pr2 qr ) ) ( hzlh ( pr2 qr ) ( nattohz ( m ) ) ) ) ) ). Proof. intro. intro. induction n. intros. split with
( dirprodpair 0 0 ). split. simpl. rewrite ( rngrunax1 hz ). rewrite
( rngrmult0 hz ). rewrite nattohzand0. change ( 0 ~> 0%hz ). apply
idpath. split. apply isreflhzieh. assumption.

intro p. set ( q' := pr1 ( pr1 ( IHn p ) ) ). set ( r' := pr2 ( pr1 ( IHn p ) ) ). set ( f := pr1 ( pr2 ( IHn p ) ) ). assert ( hzleh ( r' + 1 ) ( nattohz m + 1 ) ) as p''. apply hzlehandplusr. apply ( pr2 ( pr2 ( pr2 ( IHn p ) ) ) ). apply hzlsntoleh. assumption. set ( choice := hzlechoice ( r' + 1 ) ( nattohz m ) p' ). destruct
choice as [ k | h ]. split with ( dirprodpair q' ( r' + 1 )
). split. rewrite ( nattohzandS _ ). rewrite hzpluscomm. rewrite
f. change ( nattohz m * q' + r' + 1 ~> ( nattohz m * q' + ( r' + 1
) ) ). apply rngrassoc1. split. apply ( istranshzleh 0 r' ( r' + 1
) ). apply ( ( pr2 ( pr2 ( IHn p ) ) ) ). apply hzlhialeh. apply
hzlthnsn. assumption. split with ( dirprodpair ( q' + 1 ) 0
). split. rewrite ( nattohzandS _ ). rewrite hzpluscomm. rewrite
f. change ( nattohz m * q' + r' + 1 ~> ( nattohz m * ( q' + 1 ) +
0 ) ). rewrite ( hzplusassoc ). rewrite h. rewrite ( rngrdistr
q' _ ). rewrite rngrunax2. rewrite hzplusr0. apply idpath.
split. apply isreflhzieh. assumption. Defined.

(* A test of the division algorithm for non-negative integers: Lemma
testlemma1 : ( hzneq 0 ( 1 ) ). Proof. change 0 with ( nattohz 0%nat
). rewrite <- nattohzand1. apply nattohzandneq. intro f. apply
isreflhnat1. assert ( natlh 0 1 ) as i. apply
natlhnsn. rewrite <- f in *. assumption. Defined.

```

Lemma testlemma2 : ($\text{hzneq } 0 (1 + 1)$). Proof. change 0 with ($\text{nattohz } 0\text{nat}$). rewrite $\leftarrow \text{nattohzand1}$. rewrite $\leftarrow \text{nattohzandplus}$. apply nattohzandneg . assert ($\text{natneq } (1 + 1) 0$) as x. apply ($\text{natgthtioneq } (1 + 1) 0$). simpl. auto. intro f. apply x. apply pathsinv0 . assumption. Defined.

Lemma testlemma21 : $\text{hzlth } 0 (\text{nattohz } 2)$. Proof. change 0 with ($\text{nattohz } 0\text{nat}$). apply nattohzandlth . apply ($\text{istransnatlth } _- 1$). apply natlthnsn . apply natlthnsn . Defined.

Lemma testlemma3 : $\text{hzlth } 0 (\text{nattohz } 3)$. Proof. apply ($\text{istranshzlth } _- (\text{nattohz } 2)$). apply testlemma21. change 0 with ($\text{nattohz } 0\text{nat}$). apply nattohzandlth . apply natlthnsn . Defined.

Lemma testlemma9 : $\text{hzlth } 0 (\text{nattohz } 9)$. Proof. apply ($\text{istranshzlth } _- (\text{nattohz } 3)$). apply testlemma3. apply ($\text{istranshzlth } _- (\text{nattohz } 6)$). apply testlemma3. apply testlemma3. Defined.

Eval lazy in $\text{hzabsval} (\text{pr1} (\text{pr1} (\text{divalgorithmnnonneg } 1 (1 + 1) \text{ testlemma21 })))$. Eval lazy in $\text{hzabsval} (\text{pr1} (\text{pr1} (\text{divalgorithmnnonneg } 5) (1 + 1) \text{ testlemma21 }))$. Eval lazy in $\text{hzabsval} (\text{pr2} (\text{pr1} (\text{divalgorithmnnonneg } 5) (1 + 1) \text{ testlemma21 }))$. Eval lazy in $\text{hzabsval} (\text{pr1} (\text{pr1} (\text{divalgorithmnnonneg } 16 3 \text{ testlemma3 })))$. Eval lazy in $\text{hzabsval} (\text{pr2} (\text{pr1} (\text{divalgorithmnnonneg } 16 3 \text{ testlemma3 })))$. Eval lazy in $\text{hzabsval} (\text{pr1} (\text{pr1} (\text{divalgorithmnnonneg } 18 9 \text{ testlemma9 })))$. Eval lazy in $\text{hzabsval} (\text{pr2} (\text{pr1} (\text{divalgorithmnnonneg } 18 9 \text{ testlemma9 })))$. *

Theorem divalgorithmaxists ($n m : \text{hz}$) ($p : \text{hzneq } 0 m$) : total2 (fun qr : $\text{dirprod } hz hz \Rightarrow ((\text{dirprod } (n \rightarrow (m * (\text{pri } qr)) + (pr2 qr))) (\text{dirprod } (\text{hzleb } 0 (pr2 qr)) (\text{hzlth } (pr2 qr) (\text{nattohz } (\text{hzabsval } m))))))$). Proof. intros. destruct ($\text{hzlthoregh } n 0$) as [n_{neg} | n_{nonneg}]. destruct ($\text{hzlthoregh } m 0$) as [m_{neg} | m_{nonneg}].

(*Case I: $n < 0, m < 0$ *) set ($n' := \text{hzabsval } n$). set ($m' := \text{hzabsval } m$). assert ($\text{nattohz } m' \sim (- m)$) as f. apply hzabsvallth0 . assumption. assert ($- n \sim - (\text{nattohz } n')$) as f0. rewrite $\leftarrow (\text{hzabsvallth0 } n_{\text{neg}})$. rewrite ($\text{hzabsvallth0 } n_{\text{neg}}$). unfold n' . rewrite ($\text{hzabsvallth0 } n_{\text{neg}}$). apply idpath. assert ($\text{hzlth } 0 (\text{nattohz } m')$) as p'. assert ($\text{hzlth } 0 (- m)$) as q. apply hzlth0andminus . assumption. rewrite f. assumption. set ($a := \text{divalgorithmnnonneg } n' m' p'$). set ($q := \text{pr1} (\text{pri } a)$). set ($r := \text{pr2} (\text{pri } a)$). set ($Q := q + 1$). set ($R := - m - r$).

destruct ($\text{hzlehchoice } 0 r (\text{pr1} (\text{pr2} (\text{pr2 } a)))$) as [less | equal]. split with ($\text{dirprodpair } Q R$). split.

rewrite ($\text{pathsinv0} (\text{rngminusminus } hz n)$). assert ($- \text{nattohz } n' \sim (m * Q + R)$) as f1. unfold Q. unfold R. rewrite ($\text{pr1} (\text{pr2 } a)$). change ($\text{pr1} (\text{pri } a)$) with q. change ($\text{pr2} (\text{pri } a)$) with r. rewrite hzaddinvplus . rewrite $\leftarrow (\text{rnglmultminus } hz)$. rewrite f. rewrite (rngminusminus). rewrite ($\text{rngladdir } - q$). rewrite (hzmultl1). change ($(m * q) + - r \sim (m * q + m) + (- m + - r)$). rewrite (hzplusassoc). rewrite $\leftarrow (\text{hzplusassoc } m - -)$. change ($m + - m$) with ($m - m$). rewrite (hzrminus). rewrite (hzplusl0). apply idpath. exact ($\text{pathscomp0 } f0 f1$). split. unfold R. assert ($\text{hzlth } r (- m)$) as u. rewrite $\leftarrow (\text{hzabsvalleh0})$. apply ($\text{pr2} (\text{pr2} (\text{pr2 } a))$). apply hzltholeh . assumption. rewrite $\leftarrow (\text{hzlminus } m)$. change ($\text{pr2} (\text{dirprodpair } Q (- m - r))$) with ($- m - r$). apply hzlehandplusl . apply hzltholeh . rewrite $\leftarrow (\text{rngminusminus } hz m)$. apply hzlthminusswap . assumption. unfold R. unfold m'. rewrite hzabsvalleh0 . change ($\text{hzlth } (- m + - r) (- m)$). assert ($\text{hzlth } (- m - r) (- m + 0)$) as u. apply hzlthandplusl . apply

hzgth0andminus . apply less. assert ($- m + 0 \sim (- m)$) as f'. apply hzplusr0 . exact ($\text{transportf } (\text{fun } x : _- \Rightarrow \text{hzlth } (- m + - r) x) f' u$). apply hzltholeh . assumption. split with ($\text{dirprodpair } q 0$). split. rewrite $\leftarrow (\text{rngminusminus } hz n)$. assert ($- \text{nattohz } n' \sim (m * q + 0)$) as f1. rewrite ($\text{pr1} (\text{pr2 } a)$). change ($\text{pr1} (\text{pri } a)$) with q. change ($\text{pr2} (\text{pri } a)$) with r. rewrite hzplusr0 . rewrite (pathsinv0 equal). rewrite (hzplusr0). assert ($- (- \text{nattohz } m' * q) \sim ((- \text{nattohz } m') * q)$) as f2. apply pathsinv0 . apply rnglmultminus . rewrite f2. unfold m'. rewrite hzabsvalleh0 . apply ($\text{maponpaths } (\text{fun } x : _- \Rightarrow x * q)$). apply rngminusminus . apply hzltholeh . assumption. exact ($\text{pathscomp0 } f0 f1$). split. change ($\text{pr2} (\text{dirprodpair } q 0)$) with 0. apply (isreflholeh). rewrite equal. change ($\text{pr2} (\text{dirprodpair } q r)$) with r. apply ($\text{pr2} (\text{pr2 } a)$). destruct ($\text{hzgehchoice } m 0 m_{\text{nonneg}}$) as [h | k].

(*=====*)

(*Case II: $n < 0, m > 0$.*)

assert ($\text{hzlth } 0 (\text{nattohz } (\text{hzabsval } m))$) as p'. rewrite (hzabvalgh0). apply h. assumption. set ($a := \text{divalgorithmnnonneg } (\text{hzabsval } n) (\text{hzabsval } m) p'$). set ($q' := \text{pr1} (\text{pri } a)$). set ($r' := \text{pr2} (\text{pri } a)$). assert ($n \sim - n$) as f0. apply pathsinv0 . apply rngminusminus . assert ($- n \sim - (\text{nattohz } (\text{hzabsval } m))$) as f1. apply pathsinv0 . apply maponpaths . apply (hzabsvalleh0). apply hzltholeh . assumption. destruct ($\text{hzlchoice } 0 r' (\text{pr1} (\text{pr2} (\text{pr2 } a)))$) as [less | equal]. split with ($\text{dirprodpair } (- q' - 1) (- r')$). split. change ($\text{pr1} (\text{dirprodpair } (- q' - 1) (m - r'))$) with ($- q' - 1$). change ($\text{pr2} (\text{dirprodpair } (- q' - 1) (m - r'))$) with ($m - r'$). change ($- q' - 1$) with ($- q' + (- 1/hz)$). rewrite hzlhdistr . assert ($- \text{nattohz } (\text{hzabsval } n) \sim ((m * (- q') + m * (- 1/hz)) + (m - r'))$) as f2. rewrite ($\text{pr1} (\text{pr2 } a)$). change ($\text{pr1} (\text{pri } a)$) with q'. change ($\text{pr2} (\text{pri } a)$) with r'. rewrite hzabvalgh0 . rewrite hzaddinvplus . rewrite ($\text{rnglmultminus } hz$). rewrite ($\text{hzplusassoc } (_- (m * (- 1/hz))_-)$). apply ($\text{maponpaths } (\text{fun } u : _- \Rightarrow - (m * q') + x)$). assert ($- m + (m - r') \sim (m * (- 1/hz)) + (m - r')$) as f3. apply ($\text{maponpaths } (\text{fun } x : _- \Rightarrow x + (m - r'))$). apply pathsinv0 . assert ($m * (- 1/hz) \sim (- (m * 1/hz))$) as f30. apply ($\text{rnglmultminus } 0$). assert ($- (m * 1/hz) \sim - m$) as f31. rewrite hzmultr1 . apply idpath. rewrite f30. assumption. assert ($- r' \sim (- m + (m - r'))$) as f4. change ($- r' \sim (- m + (m + - r'))$). rewrite $\leftarrow (\text{hzplusassoc } _-)$. rewrite ($\text{hzlminus } 0$), (hzplusl0). apply idpath. rewrite f4. assumption. assumption. rewrite f0. f1. assumption.

split. change ($\text{pr2} (\text{dirprodpair } (- q' - 1) (m - r'))$) with ($m - r'$). apply hzlhandplusl . rewrite $\leftarrow (\text{hzrminus } r')$. apply hzlhandplusr . rewrite $\leftarrow (\text{hzabsvalgeh0 } m_{\text{nonneg}})$. apply ($\text{pr2} (\text{pr2 } a)$). rewrite ($\text{hzabsvalgeh0 } m_{\text{nonneg}}$). assert ($\text{hzlth } (m - r') (m + 0)$) as u. apply (hzlhandplusl). apply (hzgth0andminus). apply less. rewrite hzplusr0 in u. assumption.

split with ($\text{dirprodpair } (- q') 0$). split. change ($\text{pr1} (\text{dirprodpair } (- q') 0)$) with ($- q'$). change ($\text{pr2} (\text{dirprodpair } (- q') 0)$) with 0. assert ($- \text{nattohz } (\text{hzabsval } n) \sim (m * - q' + 0)$) as f2. rewrite (hzplusr0). rewrite ($\text{pr1} (\text{pr2 } a)$). change ($\text{pr1} (\text{pri } a)$) with q'. change ($\text{pr2} (\text{pri } a)$) with r'. rewrite $\leftarrow \text{equal}$. rewrite hzplusr0 . rewrite hzabsvalgeh0 . apply pathsinv0 . apply

```

rngrmultminus. assumption. rewrite f0.
f1. assumption. split. apply ( isreflhleh ). rewrite equal. apply
( pr2 ( pr2 ( pr2 ( a ) ) ) ).

assert empty. apply p. apply pathsinv0.
assumption. contradiction.

set ( choice2 := hzlthorgeh m 0 ). destruct choice2 as [ m_neg |
m_nonneg ].

(*Case III. Assume n>=0, m<0*) assert ( hzlth 0 ( nattohz ( hzabsval
m ) ) as p'. rewrite ( hzabsvalth0 ). rewrite <- (
rngminusminus hm m ) in m.neg. set ( d:= hzlth0andminus m.neg
). rewrite rngminusminus in d. apply d. assumption. set ( a :=
divalgorithmnonneg ( hzabsval n ) ( hzabsval m ) p' ). set ( q' :=
pri ( pri a ) ). set ( r' := pr2 ( pri a ) ). split with (
dirprodpair ( - q' ) r' ). split. rewrite <- ( hzabsvalgeh0 ).
rewrite ( pri ( pr2 ( a ) ) ). change ( pri ( pri a ) ) with
q'. change ( pr2 ( pri a ) ) with r'. change ( pri ( dirprodpair
( - q' ) r' ) ) with ( - q' ). change ( pr2 ( dirprodpair ( - q'
) r' ) ) with r'. rewrite ( hzabsvalleh0 ). apply ( maponpaths
fun x : _ => x + r' ). assert ( - m * q' > - ( m * q' ) ) as
f0. apply rnglmultminus. assert ( - ( m * q' ) > m * ( - q' ) )
as f1. apply pathsinv0. apply rngrmultminus. exact ( pathscomp0 f0
f1 ). apply hzlthtolleh. assumption. assumption. split. apply (
pri ( pr2 ( pr2 ( a ) ) ) ). apply ( pr2 ( pr2 ( pr2 ( a ) ) ) ).

(*Case IV: n>=0, m>0*)
assert ( hzlth 0 ( nattohz ( hzabsval m ) ) ) as p'. rewrite (
hzabsvalgeh0 ). destruct ( hzneqchoice 0 m ) as [ l | r ]. apply
p. assert empty. apply ( isirreflhzgth0 ). apply ( hzgthgehtrans
0 m 0 ). assumption. assumption. contradiction.

assumption. assumption. set ( a := divalgorithmnonneg ( hzabsval
n ) ( hzabsval m ) p' ). set ( q' := pri ( pri a ) ). set ( r' :=
pr2 ( pri a ) ). split with ( dirprodpair q' r' ). split.
rewrite <- hzabsvalgeh0. rewrite ( pri ( pr2 ( a ) ) ). change (
pri ( pri a ) ) with q'. change ( pr2 ( pri a ) ) with r'. change
( pri ( dirprodpair q' r' ) ) with q'. change ( pr2 ( dirprodpair
q' r' ) ) with r'. rewrite hzabsvalgeh0. apply
idpath. assumption. assumption. split. apply ( pri ( pr2 ( pr2 (
a ) ) ) ). apply ( pr2 ( pr2 ( pr2 ( a ) ) ) ). Defined.

Lemma hzdivhabsval ( a b : hz ) ( p : hzdiv a b ) : hdisj ( natleh (
hzabsval a ) ( hzabsval b ) ) ( hzabsval b > 0%nat ). Proof.
intros a b p P q. apply ( p P ). intro t. destruct t as [ k f ]. unfold
hzdiv0 in f. apply q. apply natdivieh with ( hzabsval k ). rewrite (
hzabsvalandmult ). rewrite f. apply idpath. Defined.

Lemma divalgorithm ( n m : hz ) ( p : hzneq 0 m ) : iscontr ( total2 (
fun qr : hzprod hz qr => ( ( dirprod ( n > ( ( m * ( pri qr ) ) +
( pr2 qr ) ) ) ( dirprod ( hzleh 0 ( pr2 qr ) ) ( hzlth ( pr2 qr ) (
nattohz ( hzabsval m ) ) ) ) ) ) ). Proof.
intros. split with ( divalgorithmexists n m p ). intro t. destruct t as [ qr' t' ]. destruct qr' as [ qr' r' ]. simpl in t'. destruct t' as [ f' p2p2t ]. destruct p2p2t as [ pip2p2t p2p2p2t ]. destruct divalgorithmexists
as [ qr v ]. destruct qr as [ qr ]. destruct v as [ f p2p2dae ]. destruct
p2p2dae as [ pip2p2dae p2p2p2dae ]. simpl in f. simpl in
pip2p2dae. simpl in p2p2p2dae.

assert ( r' > r ) as h. (*Proof that r' > r :*) assert ( m * ( q
- q' ) > ( r' - r ) ) as h0. change ( q - q' ) with ( q + - q' )
. rewrite ( hzldistr ). rewrite <- ( hzplusr0 ( r' - r ) )
. rewrite <- ( hzrminus ( m * q' ) ). change ( r' - r ) with ( r'
+ ( - r ) ). rewrite ( hzplusassoc r' ). change ( ( m * q' ) - ( m
* q' ) ) with ( ( m * q' ) + ( - ( m * q' ) ) ). rewrite <- (
hzplusassoc ( - r ) ). rewrite ( hzpluscomm ( - r ) ). rewrite <- (
hzplusassoc r' ). rewrite <- ( hzplusassoc r' ). rewrite (
hzpluscomm r' ). rewrite <- f'. rewrite f. rewrite ( hzplusassoc (
m * q ) ). change ( r + - r ) with ( r - r ). rewrite ( hzrminus ). rewrite
( hzplusr0 ). rewrite ( rngrmultminus hz ). change ( m * q
+ - ( m * q' ) ) with ( ( m * q + - ( m * q' ) ) %rng ). apply
idpath.

assert ( hdisj ( natleh ( hzabsval m ) ( hzabsval ( r' - r ) ) ) (
hzabsval ( r' - r ) > 0%nat ) ) as v. apply hzdivhabsval. intro
P. intro s. apply s. split with ( q - q' ). unfold
hzdiv0. assumption. assert ( isaprop ( r' > r ) ) as P. apply
isasethz. apply ( v ( hProppair ( r' > r ) P ) ). intro
s. destruct s as [ left | right ]. assert ( hzlth ( nattohz (
hzabsval ( r' - r ) ) ) ( nattohz ( hzabsval m ) ) ) as u.
destruct ( hzgthorleh r' r ) as [ greater | less ]. assert (
hzlth 0 ( r' - r ) ) as e. rewrite <- ( hzrminus r ). apply
hzlthandplusr. assumption. rewrite ( hzabsvalgth0 ). apply
hzlthminus. apply ( p2p2p2t ). apply ( p2p2p2dae ). apply
( p1p2p2dae ). apply e. destruct ( hzlechoice r' r lesseq ) as [
less | equal ]. rewrite hzabsvalandminuspos. rewrite
hzabsvalgth0. apply hzlthminus. apply ( p2p2p2dae ). apply
( p2p2p2t ). apply ( p1p2p2t ). apply hz1lthminusequiv.
assumption. apply ( p1p2p2t ). apply p1p2p2dae. rewrite
equal. rewrite hzrminus. rewrite hzabsval0. rewrite
nattohzand0. apply hzabsvalneq0. intro Q. apply p. assumption.
assert empty. apply ( isirreflhzlh ( nattohz ( hzabsval m ) ) ). apply
( hzlelhtrans _ ( nattohz ( hzabsval ( r' - r ) ) ) _ ). apply
nattohzandleh. assumption. assumption. contradiction.
assert ( r' > r ) as i. assert ( r' - r > 0 ) as i0. apply
hzabsvalneq0. assumption. rewrite <- ( hzplusl0 r ). rewrite <- ( hzplusr0 r' ). assert ( r' + ( r - r ) > ( 0 + r ) ) as i00.
change ( r - r ) with ( r + - r ). rewrite ( hzpluscomm _ ( - r ) )
. rewrite <- hzplusassoc. apply ( maponpaths ( fun x : _ => x +
r ) ). apply i0. exact ( transportf ( fun x : _ => ( r' + x ) >
( 0 + r ) ) ) ( ( hzrminus r ) ) i00. apply i.

assert ( q' > q ) as g. (*Proof that q' > q :*) rewrite h in
f'. rewrite f in f'. apply ( hzmultcan q' q m ). intro i. apply
p. apply pathsinv0. assumption. apply ( hzplusrcan ( m * q' ) ( m *
q ) r ). apply pathsinv0. apply f'.

(* Path in direct product: *) assert ( dirprodpair q' r' > (
dirprodpair q r ) ) as j. apply
pathsdirprod. assumption. assumption.

(* Proof of general path: *) apply pathintotalfiber with ( p0 := j
). assert ( iscontr ( dirprod ( n > ( m * q + r ) ) ( dirprod (
hzleh 0 r ) ( hzlth r ( nattohz ( hzabsval m ) ) ) ) ) ) as
contract. change iscontr with ( isofhlevel0 0 ). apply
isofhleveldirprod. split with f. intro t. apply isasethz. apply
isofhleveldirprod. split with pip2p2dae. intro t. apply hzleh.
split with p2p2p2dae. intro t. apply hz1lth. apply
proofirrelevancecontr. assumption. Defined.

Definition hzquotientmod ( p : hz ) ( x : hzneq 0 p ) : hz -> hz := fun n : hz => ( pr1 ( pr1 ( divalgorithmexists n p x ) ) ).

Definition hzremaindermod ( p : hz ) ( x : hzneq 0 p ) : hz -> hz := fun n : hz => ( pr2 ( pr1 ( divalgorithmexists n p x ) ) ).

Definition hzdivequationmod ( p : hz ) ( x : hzneq 0 p ) ( n : hz ) : n > ( p * ( hzquotientmod p x n ) + ( hzremaindermod p x n ) ) := ( pr1 ( pr2 ( divalgorithmexists n p x ) ) ).

Definition hzleh0remaindermod ( p : hz ) ( x : hzneq 0 p ) ( n : hz )

```

```

: hzleh 0 ( hzremaindermod p x n ) := ( pr1 ( pr2 ( pr2 (
divalgorithmxists n p x ) ) ) ). Defined.

Definition hzthremaindermod ( p : hz ) ( x : hzneq 0 p ) ( n : hz
) : hzlh ( hzremaindermod p x n ) ( nattohz ( hzabsval p ) ) := ( pr2
( pr2 ( pr2 ( divalgorithmxists n p x ) ) ) ).

(* Eval lazy in hzabsval ( ( ( hzquotientmod ( 1 + 1 ) testlemma2 ( 1
+ 1 + 1 + 1 + 1 + 1 + 1 ) ) ) ). Eval lazy in hzabsval ( ( (
hzremaindermod ( 1 + 1 ) testlemma2 ( 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 ) ) ) ). *)

(** * II. QUOTIENTS AND REMAINDERS *)

Definition isaprime ( p : hz ) : UU := dirprod ( hzlth 1 p ) ( forall
m : hz, ( hzdiv m p ) -> ( hdisj ( m ~> 1 ) ( m ~> p ) ) ).

Lemma isapropisaprime ( p : hz ) : isaprop ( isaprime p ). Proof.
intros. apply isofhleveldirprod. apply ( hzlth 1 p ). apply
impred. intro m. apply impredfun. apply ( hdisj ( m ~> 1 ) ( m ~> p )
). Defined.

Lemma isaprimeoneq0 { p : hz } ( x : isaprime p ) : hzneq 0 p.
Proof. intros. intros f. apply ( isirreflhzh0 ). apply (
istranshzlh _ _ ). apply hzlhnsn. rewrite f. apply ( pr1 x ). Defined.

Lemma hzqrtest ( m : hz ) ( x : hzneq 0 m ) ( a q r : hz ) : dirprod (
a ~> ( ( m * q ) + r ) ) ( dirprod ( hzleh 0 r ) ( hzlh r ( nattohz
(hzabsval m ) ) ) ) -> dirprod ( q ~> hzquotientmod m x a ) ( r ~>
hzremaindermod m x a ). Proof. intros m x a q r d. set ( k := tpair
( P := ( fun qr : dirprod hz hz -> dirprod ( a ~> ( m * ( pri 1 p ) +
pr2 qr ) ) ( dirprod ( hzleh 0 ( pr2 qr ) ) ( hzlh ( pr2 qr ) (
nattohz ( hzabsval m ) ) ) ) ) ( dirprodpair q r ) d ). assert ( k
~> ( pr1 ( divalgorith a m x ) ) ) as f. apply ( pr2 ( divalgorith
a x ) ). split. change q with ( pr1 ( pri 1 k ) ). rewrite f. apply
idpath. change r with ( pr2 ( pri 1 k ) ). rewrite f. apply idpath.
Defined.

Definition hzqrtestq ( m : hz ) ( x : hzneq 0 m ) ( a q r : hz ) ( d :
dirprod ( a ~> ( ( m * q ) + r ) ) ( dirprod ( hzleh 0 r ) ( hzlh r (
nattohz ( hzabsval m ) ) ) ) ) := pr1 ( hzqrtest m x a q r d ).

Definition hzqrtestr ( m : hz ) ( x : hzneq 0 m ) ( a q r : hz ) ( d :
dirprod ( a ~> ( ( m * q ) + r ) ) ( dirprod ( hzleh 0 r ) ( hzlh r (
nattohz ( hzabsval m ) ) ) ) ) := pr2 ( hzqrtest m x a q r d ).

Lemma hzrand0eq ( p : hz ) ( x : hzneq 0 p ) : 0 ~> ( ( p * 0 ) + 0
). Proof. intros. rewrite hzmultx0. rewrite hzplusl0. apply idpath.
Defined.

Lemma hzrand0ineq ( p : hz ) ( x : hzneq 0 p ) : dirprod ( hzleh 0 0
) ( hzlh 0 ( nattohz ( hzabsval p ) ) ). Proof.
intros. split. apply isreflhzhle. apply hzabsvalneq0. assumption.
Defined.

Lemma hzrand0q ( p : hz ) ( x : hzneq 0 p ) : hzquotientmod p x 0 ~>
0. Proof. intros. apply pathsinv0. apply ( hzqrtestq p x 0 0 0 ). split.
apply ( hzrand0eq p x ). apply ( hzrand0ineq p x ). Defined.

Lemma hzrand0r ( p : hz ) ( x : hzneq 0 p ) : hzremaindermod p x 0 ~>
0. Proof. intros. apply pathsinv0. apply ( hzqrtestr p x 0 0 0 ). split.
apply ( hzrand0eq p x ). apply ( hzrand0ineq p x ). Defined.

Lemma hzrandied ( p : hz ) ( is : isaprime p ) : 1 ~> ( ( p * 0 ) + 1
). Proof. intros. rewrite hzmultx0. rewrite hzplusl0. apply idpath.

```

)). Defined.

Lemma hzrandtimeseq (m : hz) (x : hzneq 0 m) (a b : hz) : (a * b) \sim > ((m * ((hzquotientmod m x) a * (hzquotientmod m x) b * m + (hzremaindermod m x b) * (hzquotientmod m x a) + (hzremaindermod m x a) * (hzquotientmod m x b) + (hzquotientmod m x (hzremaindermod m x a * hzremaindermod m x b)))) + hzremaindermod m x (hzremaindermod m x a * hzremaindermod m x b)). Proof. intros. rewrite 3! (hzldistr). rewrite (hzplusassoc _ _ (hzremaindermod m x (hzremaindermod m x a * hzremaindermod m x b))). rewrite <- hzdivequationmod. rewrite (hzmultassoc _ _ m). rewrite <- (hzmultcomm _ m). change ((m * hzquotientmod m x a * (m * hzquotientmod m x b))\hz + m * (hzremaindermod m x b * hzquotientmod m x a))\rung with ((m * hzquotientmod m x a * (m * hzquotientmod m x b)) + m * (hzremaindermod m x b * hzquotientmod m x a))\hz. change (a * b) > ((m * hzquotientmod m x a * (m * hzquotientmod m x b) + m * (hzremaindermod m x b * hzquotientmod m x a))\hz + m * (hzremaindermod m x a * hzquotientmod m x b))\hz\rung + hzremaindermod m x a * hzremaindermod m x b with (a * b \sim > ((m * hzquotientmod m x a * (m * hzquotientmod m x b) + m * (hzremaindermod m x b * hzquotientmod m x a)) + m * (hzremaindermod m x a * hzquotientmod m x b))\hz + hzremaindermod m x a * hzremaindermod m x b). rewrite (hzplusassoc (m * hzquotientmod m x a * (m * hzquotientmod m x b)))). rewrite (hzmultcomm (m * (hzremaindermod m x b * hzquotientmod m x a))) (m * (hzremaindermod m x a * hzquotientmod m x b))). rewrite <- (hzmultassoc (m * (hzremaindermod m x a * (m * hzquotientmod m x b))))). rewrite (hzmultassoc (hzremaindermod m x a) m (hzquotientmod m x b)). rewrite <- (hzplusassoc (m * hzquotientmod m x a * (m * hzquotientmod m x b))). rewrite <- (hzdivequationmod. rewrite hzplusassoc. rewrite (hzmultcomm (hzremaindermod m x b) (hzquotientmod m x a)). rewrite <- (hzmultcomm (hzremaindermod m x a) (hzquotientmod m x b)). rewrite (hzmultassoc (hzremaindermod m x a) m (hzquotientmod m x b)). rewrite <- (hzldistr). rewrite <- hzdivequationmod. rewrite <- (hzrdistr). rewrite <- hzdivequationmod. rewrite hzplusassoc. rewrite (hzmultcomm (hzremaindermod m x b) (hzquotientmod m x a)). rewrite <- (hzmultassoc (m * (hzremaindermod m x a * hzquotientmod m x b)))). Proof. intros. split. apply hzlehOremaindermod. apply hzlthremaindermod. Defined.

Lemma hzrandtimesineq (m : hz) (x : hzneq 0 m) (a b : hz) : dirprod (hzleh 0 (hzremaindermod m x (hzremaindermod m x a * hzremaindermod m x b))) (hzith (hzremaindermod m x a * hzremaindermod m x b)) (nattohz (hzabsval m)). Proof. intros. split. apply hzlehOremaindermod. apply hzlthremaindermod. Defined.

Lemma hzquotientmodandtimes (m : hz) (x : hzneq 0 m) (a b : hz) : hzquotientmod (a * b) \sim > (hzquotientmod m x) a * (hzquotientmod m x) b * m + (hzremaindermod m x b) * (hzquotientmod m x a) + (hzremaindermod m x a) * (hzquotientmod m x b) + (hzquotientmod m x (hzremaindermod m x a * hzremaindermod m x b))). Proof. intros. apply pathsinv0. apply (hzqtestq m x (a * b) - (hzremaindermod m x (hzremaindermod m x a * hzremaindermod m x b))). split. apply hzrandtimeseq. apply hzrandtimesineq. Defined.

Lemma hzremaindermodandtimes (m : hz) (x : hzneq 0 m) (a b : hz) : hzremaindermod m x (a * b) \sim > (hzremaindermod m x (hzremaindermod m x a * hzremaindermod m x b)). Proof. intros. apply pathsinv0. apply (hzqtestr m x (a * b) ((hzquotientmod m x) a * (hzquotientmod m x) b * m + (hzremaindermod m x b) * (hzquotientmod m x a) + (hzremaindermod m x a) * (hzquotientmod m x b) + (hzquotientmod m x (hzremaindermod m x a * hzremaindermod m x b)))). split. apply hzrandtimeseq. apply hzrandtimesineq. Defined.

Lemma hzrandremaindereq (m : hz) (is : hzneq 0 m) (n : hz) : (

hzremaindermod m is n \sim > ((m * (pr1 (dirprodpair 0 (hzremaindermod m is n))) + (pr2 (dirprodpair (@rngunel1 hz) (hzremaindermod m is n))))). Proof. intros. simpl. rewrite hzmultx0. rewrite hzplusl0. apply idpath. Defined.

Lemma hzrandremainderineq (m : hz) (is : hzneq 0 m) (n : hz) : dirprod (hzleh (@rngunel1 hz) (hzremaindermod m is n)) (hzith (hzremaindermod m is n) (nattohz (hzabsval m))). Proof. intros. split. apply hzlehOremaindermod. apply hzlthremaindermodmod. Defined.

Lemma hzremaindermoditerated (m : hz) (is : hzneq 0 m) (n : hz) : hzremaindermod m is (hzremaindermod m is n) \sim > (hzremaindermod m is n). Proof. intros. apply pathsinv0. apply (hzqtestr m is (hzremaindermod m is n) 0 (hzremaindermod m is n)). split. apply hzrandremaindereq. apply hzrandremainderineq. Defined.

Lemma hzrandremainderq (m : hz) (is : hzneq 0 m) (n : hz) : 0 \sim > hzquotientmod m is (hzremaindermod m is n). Proof. intros. apply (hzqtestq m is (hzremaindermod m is n) 0 (hzremaindermod m is n)). split. apply hzrandremaindereq. apply hzrandremainderineq. Defined.

(** * III. THE EUCLIDEAN ALGORITHM *)

Definition iscommonhzdiv (k n m : hz) := dirprod (hzdiv k n) (hzdiv k m).

Lemma isapropiscommonhzdiv (k n m : hz) : isaprop (iscommonhzdiv k n m). Proof. intros. unfold isaprop. apply isofhlevekdirprod. apply hzdiv. apply hzdiv. Defined.

Definition hzgcd (n m : hz) : UU := total2 (fun k : hz => dirprod (iscommonhzdiv k n m) (forall l : hz, iscommonhzdiv l n m \rightarrow hzleh l k)).

Lemma isaprophzgcd0 (k n m : hz) : isaprop (dirprod (iscommonhzdiv k n m) (forall l : hz, iscommonhzdiv l n m \rightarrow hzleh l k)). Proof. intros. apply isofhlevekdirprod. apply isapropiscommonhzdiv. apply impred. intro t. apply impredfun. apply hzleh. Defined.

Lemma isaprophzgcd (n m : hz) : isaprop (hzgcd n m). Proof. intros. intros k l. assert (isofhlevel 2 (hzgcd n m)) as aux. apply isofhleveletal2. apply isasethz. intros x. apply hleveletns. apply isofhlevekdirprod. apply isapropiscommonhzdiv. apply impred. intro t. apply impredfun. apply (hzleh t x). assert (k \sim > l) as f. destruct k as [k pq]. destruct pq as [p q]. destruct l as [l pq]. destruct pq as [p' q']. assert (k \sim > l) as f0. apply isantisymmhzleh. apply q'. assumption. apply q. assumption.

apply pathintotalfiber with (p0 := f0). assert (isaprop (dirprod (iscommonhzdiv l n m) (forall x : hz, iscommonhzdiv x n m \rightarrow hzleh x l))) as is. apply isofhlevekdirprod. apply isapropiscommonhzdiv. apply impred. intro t. apply impredfun. apply (hzleh t l). apply is. split with f. intro g. destruct k as [k pq]. destruct pq as [p q]. destruct l as [l pq]. destruct pq as [p' q']. apply aux. Defined.

(* Euclidean algorithm for calculating the GCD of two numbers (here assumed to be natural numbers (m \leq n)):

gcd (n , m) := 1. if m = 0, then take n. 2. if m \neq 0, then divide n = q * m + r and take g := gcd (m , r). *)

Lemma hzdivandmultl (a c d : hz) (p : hzdiv d a) : hzdiv d (c * a). Proof. intros. intros P s. apply p. intro k. destruct k as [k f

```
j. apply s. unfold hzdiv0. split with ( c * k ). rewrite ( hzmultcomm d ). rewrite ( hzmultassoc ). unfold hzdiv0 in f. rewrite ( hzmultcomm k ). rewrite f. apply idpath. Defined.
```

```
Lemma hzdivandmultr ( a c d : hz ) ( p : hzdiv d a ) : hzdiv d ( a * c ). Proof. intros. rewrite hzmultcomm. apply hzdivandmultr. assumption. Defined.
```

```
Lemma hzdivandminus ( a d : hz ) ( p : hzdiv d a ) : hzdiv d ( - a ). Proof. intros. intros P s. apply p. intro k. destruct k as [ k f ]. apply s. split with ( - k ). unfold hzdiv0. unfold hzdiv0 in f. rewrite ( rngrmultminus hz ). apply maponpaths. assumption. Defined.
```

```
Definition natgcd ( m n : nat ) : ( natneq 0%nat n ) -> ( natleh m n ) -> ( hzgcd ( nattohz n ) ( nattohz m ) ). Proof. set ( E := ( fun m : nat => forall n : nat, ( natneq 0%nat n ) -> ( natleh m n ) -> ( hzgcd ( nattohz n ) ( nattohz m ) ) ) ). assert ( forall x : nat, E x ) as goal. apply stronginduction. (* BASE CASE: *) intros n x0 x1. split with ( nattohz n ). split. unfold iscommonhzdiv. split. unfold hzdiv. intros P s. apply s. unfold hzdiv0. split with 1. rewrite hzmultr1. apply idpath. unfold hzdiv. intros P s. apply s. unfold hzdiv0. split with 0. rewrite hzmultr0. rewrite nattohzand0. apply idpath. intros t. destruct t as [ t0 t1 ]. destruct ( hzgthorle t0 ) as [ left | right ]. rewrite <- hzabsvalgh0. apply nattohzandleh. unfold hzdiv in t0. apply t0. intro t2. destruct t2 as [ t k t2 ]. unfold hzdiv0 in t2. assert ( coprod ( natleh ( hzabsval 1 ) n ) ( n > 0%nat ) ) as C. apply ( natdiveleh ( hzabsval 1 ) ( n ) ( hzabsval k ) ). apply ( isinclnsinj isinclnattohz ). rewrite nattohzandmult. rewrite 2! hzabsvalgeh0. assumption. assert ( hzgth ( 1 * k ) ( 1 * 0 ) ) as i. rewrite hzmultr0. rewrite t2. change 0 with ( nattohz 0%nat ). apply nattohzandgeh. apply x1. apply ( hzgemandmultinv _ _ 1 ). assumption. assumption. apply hzgthtoge. assumption. destruct C as [ C0 | C1 ]. assumption. assert empty. apply x0. apply pathsinv0. assumption. contradiction. assumption. apply (istranshzhle _ 0 _ ). assumption. change 0 with ( nattohz 0%nat ). apply nattohzandleh. assumption. (* INDUCTION CASE: *) intros m p q. intros n i j.
```

```
assert ( hzlth 0 ( nattohz m ) ) as p'. change 0 with ( nattohz 0%nat ). apply nattohzandlth. apply natneq0togh0. apply p. set ( a := divalgorithmnneg n m p' ). destruct a as [ qr a ]. destruct qr as [ quot rem ]. destruct a as [ f a ]. destruct a as [ a b ]. simpl in b. simpl in f. assert ( natlth ( hzabsval rem ) m ) as p''. rewrite <- ( hzabsvalandnattohz m ). apply nattohzandlthinv. rewrite 2! hzabsvalgeh0. assumption. apply hzgthtoge. apply ( hzgthgehtrans _ rem ). assumption. assumption. assumption. assert ( natleh ( hzabsval rem ) n ) as i''. apply natlthtoleh. apply nattohzandlthinv. rewrite hzabsvalgeh0. apply ( hzlthletrans _ ( nattohz m ) _ ). assumption. apply nattohzandleh. assumption. assumption. assert ( natneq 0%nat m ) as p'''. intro ff. apply p. apply pathsinv0. assumption. destruct ( q ( hzabsval rem ) p'' m p''' ( natlthtoleh _ _ p'') ) as [ rr c ]. destruct c as [ c0 c1 ]. split with rr. split. split. apply ( hzdivlinearcombright ( nattohz n ) ( nattohz m * quot ) ( rem ) rr f ). apply hzdivandmultr. exact ( pr1 c0 ). rewrite hzabsvalgeh0 in c0. exact ( pr2 c0 ). assumption. exact ( pr1 c0 ).
```

```
intros l o. apply c1. split. exact ( pr2 o ). rewrite hzabsvalgeh0. apply ( hzdivlinearcombleft ( nattohz n ) ( nattohz m * quot ) ( rem ) l f ). exact ( pr1 o ). apply hzdivandmultr. exact ( pr2 o ). assumption. assumption. Defined.
```

```
Lemma hzgcdandminusl ( m n : hz ) : hzgcd m n -> hzgcd ( - m ) n.
```

```
Proof. intros. assert ( hProppair ( hzgcd m n ) ( isaprophzgcd _ _ ) -> ( hProppair ( hzgcd ( - m ) n ) ( isaprophzgcd _ _ ) ) ) as x. apply uahp. intro i. destruct i as [ a i ]. destruct i as [ io ii ]. destruct io as [ jo ji ]. split with a. split. split. apply jo. intro k. destruct k as [ k f ]. unfold hzdiv0 in f. intros P s. apply s. split with ( - k ). unfold hzdiv0. rewrite ( rngrmultminus hz ). apply maponpaths. assumption. assumption. intros l f. apply ii. split. apply ( pr1 f ). intro k. destruct k as [ k g ]. unfold hzdiv0 in g. intros P s. apply s. split with ( - k ). unfold hzdiv0. rewrite ( rngrmultminus hz ). rewrite <- ( rngrminusminus hz m ). apply maponpaths. assumption. exact ( pr2 f ). intro i. destruct i as [ a i ]. destruct i as [ io ii ]. destruct io as [ jo ji ]. split with a. split. split. apply jo. intro k. destruct k as [ k f ]. unfold hzdiv0 in f. intros P s. apply s. split with ( - k ). unfold hzdiv0. rewrite ( rngrmultminus hz ). rewrite <- ( rngrminusminus hz m ). apply maponpaths. assumption. assumption. intros l f. apply ii. split. apply ( pr1 f ). intro k. destruct k as [ k g ]. unfold hzdiv0 in g. intros P s. apply s. split with ( - k ). unfold hzdiv0. rewrite ( rngrmultminus hz ). apply maponpaths. assumption. exact ( pr2 f ). apply ( pathintotalpri x ). Defined.
```

```
Lemma hzgcdsymm ( m n : hz ) : hzgcd m n -> hzgcd n m. Proof. intros. assert ( hProppair ( hzgcd m n ) ( isaprophzgcd _ _ ) -> ( hProppair ( hzgcd n m ) ( isaprophzgcd _ _ ) ) ) as x. apply uahp. intro i. destruct i as [ a i ]. destruct i as [ io ii ]. destruct io as [ jo ji ]. split with a. split. split. assumption. assumption. intros l o. apply ii. split. exact ( pr2 o ). exact ( pri o ). intro i. destruct i as [ a i ]. destruct i as [ io ii ]. destruct io as [ jo ji ]. split with a. split. split. assumption. assumption. intros l o. apply ii. split. exact ( pr2 o ). exact ( pri o ). apply ( pathintotalpri x ). Defined.
```

```
Lemma hzgcdandminusr ( m n : hz ) : hzgcd m n -> hzgcd m ( - n ). Proof. intros. rewrite 2! ( hzgcdsymm m ). rewrite hzgcdandminusl. apply idpath. Defined.
```

```
Definition euclidean ( n m : hz ) ( i : hnqe 0 n ) ( p : natleh ( hzabsval m ) ( hzabsval n ) ) : hzgcd n m. Proof. intros. assert ( natneq 0%nat ( hzabsval n ) ) as j. intro x. apply i. assert ( hzabsval n > 0%nat ) as f. apply pathsinv0. assumption. rewrite ( hzabsvaleq0 f ). apply idpath. set ( a := natgcd ( hzabsval m ) ( hzabsval n ) j p ). destruct ( hzlthorgeh 0 n ) as [ left_n | right_n ]. destruct ( hzlthorgeh 0 m ) as [ left_m | right_m ]. rewrite 2! ( hzabsvalgh0 ) in a. rewrite hzabsvalleh0 in a. rewrite hzgcdandminusr. assumption. assumption. assumption. destruct ( hzlthorgeh 0 m ) as [ left_m | right_m ]. rewrite ( hzabsvalgh0 left_m ) in a. rewrite hzabsvalleh0 in a. rewrite hzgcdandminusl. assumption. assumption. rewrite 2! hzabsvalleh0 in a. rewrite hzgcdandminusl. rewrite hzgcdandminusr. assumption. assumption. Defined.
```

```
Theorem euclideanalgorithm ( n m : hz ) ( i : hnqe 0 n ) : iscontr ( hzgcd n m ). Proof. intros. destruct ( natgthorleh ( hzabsval m ) ( hzabsval n ) ) as [ left | right ]. assert ( hnqe 0 m ) as i'. intro f. apply ( negnatlthn0 ( hzabsval n ) ). rewrite <- f in left. rewrite hzabsval0 in left. assumption. set ( a := ( euclidean m n i' ( natlthtoleh _ _ left ) ) ). rewrite hzgcdsymm in a. split with a. intro. apply isaprophzgcd. split with ( euclidean m i right ). intro. apply isaprophzgcd. Defined.
```

```
Definition gcd ( n m : hz ) ( i : hnqe 0 n ) : hz := pr1 ( pr1 ( euclideanalgorithm n m i ) ).
```

```
Definition gcdiscommdiv ( n m : hz ) ( i : hnqe 0 n ) := pr1 ( pr2
```

```

(pr1 ( euclideanalgorithm n m i ) )).

Definition gcdisgreatest ( n m : hz ) ( i : hzneq 0 n ) := pr2 ( pr2 ( pr1 ( euclideanalgorithm n m i ) ) ).

Lemma hzdivand0 ( n : hz ) : hzdiv n 0. Proof. intros. intros P s. apply s. split with 0. unfold hzdiv0. apply hzmultx0. Defined.

Lemma nozerodiv ( n : hz ) ( i : hzneq 0 n ) : neg ( hzdiv 0 n ). Proof. intros. intro p. apply i. apply ( p ( hProppair ( 0 > n ) ( isasethz 0 n ) ) ). intro t. destruct t as [ k f ]. unfold hzdiv0 in f. rewrite ( hzmultx0 ) in f. assumption. Defined.

(** * IV. Bezout's lemma and the commutative ring Z/pZ *)

Lemma commonhzdivsignswap ( k n m : hz ) ( p : iscommonhzdiv k n m ) : iscommonhzdiv ( - k ) n m . Proof. intros. destruct p as [ p0 p1 ]. split. apply p0. intro t. intros P s. apply s. destruct t as [ 1 f ]. unfold hzdiv0 in f. split with ( - 1 ). unfold hzdiv0. change ( k * 1 ) with ( k * 1 )%rng in f. rewrite <- rnmultminusminus in f. assumption. apply p1. intro t. destruct t as [ 1 f ]. unfold hzdiv0 in f. intros P s. apply s. split with ( - 1 ). unfold hzdiv0. change ( k * 1 ) with ( k * 1 )%rng in f. rewrite <- rnmultminusminus in f. assumption. Defined.

Lemma gcdneq0 ( n m : hz ) ( i : hzneq 0 n ) : hzneq 0 ( gcd n m i ). Proof. intros. intro f. apply ( nozerodiv n ). assumption. rewrite f. exact ( pr1 ( gcdiscommdiv n m i ) ). Defined.

Lemma gcdpositive ( n m : hz ) ( i : hzneq 0 n ) : hzlh0 ( gcd n m i ). Proof. intros. destruct ( hzneqchoice 0 ( gcd n m i ) ( gcdneq0 n m i ) ) as [ left | right ]. assert empty. assert ( hzleh ( - ( gcd n m i ) ) ) as i0. apply ( gcdisgreatest n m i ). apply commonhzdivsignswap. exact ( gcdiscommdiv n m i ). apply ( isirreflzh0 ). apply ( istranshzlth _ ( - ( gcd n m i ) ) _ ). apply hzlh0andminus. assumption. apply ( hzlehlhtrans _ ( gcd n m i ) _ ). assumption. assumption. contradiction. assumption. Defined.

Lemma gcdanddiv ( n m : hz ) ( i : hzneq 0 n ) ( p : hzdiv n m ) : coprod ( gcd n m i > n ) ( gcd n m i > - n ). Proof. intros. destruct ( hzneqchoice 0 n i ) as [ left | right ]. apply ii2. apply isantisymmhzleh. apply ( hzdivhzabsval ( gcd n m i ) n ( pr1 ( gcdiscommdiv n m i ) ) ). intro c'. destruct c' as [ c0 | c1 ]. rewrite <- ( hzabsvalgeh0 ). rewrite <- ( hzabsvalth0 ). apply nattohzandleh. assumption. assumption. apply hzgthtogh. apply ( gcdpositive n m i ). assert empty. assert ( n > 0 ) as f. rewrite hzabsvaleq0. apply idpath. assumption. apply i. apply pathsinv0. assumption. contradiction. apply ( pr2 ( pr2 ( pr1 ( euclideanalgorithm n m i ) ) ) ). apply commonhzdivsignswap. split. apply hzdivisrefl. assumption. apply iii. apply isantisymmhzleh. apply ( hzdivhzabsval ( gcd n m i ) n ( pr1 ( gcdiscommdiv n m i ) ) ). intro c'. destruct c' as [ c0 | c1 ]. rewrite <- hzabsvalgh0. assert ( n > nattohz ( hzabsval n ) ) as f. apply pathsinv0. apply hzabsvalgh0. assumption. assert ( hzleh ( nattohz ( hzabsval ( gcd n m i ) ) ) ( nattohz ( hzabsval n ) ) ) as j. apply nattohzandleh. assumption. exact ( transportf ( fun x : _ => hzleh ( nattohz ( hzabsval ( gcd n m i ) ) ) x ) ( pathsinv0 f ) j ). apply gcdpositive. assert empty. apply i. apply pathsinv0. rewrite hzabsvaleq0. apply idpath. assumption. contradiction. apply ( gcdisgreatest n m i ). split. apply hzdivisrefl. assumption. Defined.

Lemma gcdand0 ( n : hz ) ( i : hzneq 0 n ) : coprod ( gcd n 0 i > n ) ( gcd n 0 i > - n ). Proof. intros. apply gcdanddiv. apply hzdivand0. Defined.

Lemma natbezoutstrong ( m n : nat ) ( i : hzneq 0 ( nattohz n ) ) :
total2 ( fun ab : dirprod hz hz => ( gcd ( nattohz n ) ( nattohz m ) i > ( ( pr1 ab ) * ( nattohz n ) + ( pr2 ab ) * ( nattohz m ) ) ) ). Proof. set ( E := ( fun m : nat => forall n : nat, forall i : hzneq 0 ( nattohz n ), total2 ( fun ab : dirprod hz hz => gcd ( nattohz n ) ( nattohz m ) i > ( ( pr1 ab ) * ( nattohz n ) + ( pr2 ab ) * ( nattohz m ) ) ) ) ). assert ( forall x : nat, E x ) as goal. apply stronginduction. (* Base Case: *) unfold E. intros. split with ( dirprodpair 1 0 ). simpl. rewrite nattohzand0. destruct ( gcdand0 ( nattohz n ) i ) as [ left | right ]. rewrite hzmultl1. rewrite hzplusr0. assumption. assert empty. apply ( isirreflzh0 ( gcd ( nattohz n ) 0 i ) ). apply ( istranshzlth _ 0 _ ). rewrite right. apply hzlh0andminus. change 0 with ( nattohz 0%nat ). apply nattohzand0. apply natneq0togh0. intro f. apply i. rewrite f. apply idpath. apply gcdpositive. contradiction. (* Induction Case: *) intros m x y. intros n i. assert ( hzneq 0 ( nattohz m ) ) as p. intro f. apply x. apply pathsinv0. rewrite <- hzabsvalandnattohz. change 0%nat with ( hzabsval ( nattohz 0%nat ) ). apply maponpaths. assumption. set ( r := hzremaindermod ( nattohz m ) p ( nattohz n ) p ( nattohz m ) ). set ( q := hzquotientmod ( nattohz m ) p ( nattohz n ) ). assert ( natth ( hzabsval r ) m ) as p'. rewrite <- ( hzabsvalandnattohz m ). apply hzabsvalandlh. exact ( hzlehOremaindermod ( nattohz m ) p ( nattohz n ) ). unfold r. unfold hzremaindermod. rewrite <- ( hzabsvalgeh0 ( pr1 ( pr2 ( pr2 ( divalgorithmexists ( nattohz n ) ( nattohz m ) p ) ) ) ) ). apply nattohzand0. assert ( natth ( hzabsval ( pr2 ( pr1 ( divalgorithmexists ( nattohz n ) ( nattohz m ) p ) ) ) ) ( ( hzabsval ( nattohz m ) ) ) ) as ii. apply hzabsvalandlh. exact ( hzlehOremaindermod ( nattohz m ) p ( nattohz n ) ). assert ( nattohz ( hzabsval ( nattohz m ) ) > ( nattohz m ) ) as f. apply maponpaths. apply hzabsvalandnattohz. exact ( transportf ( fun x : _ => hzlh ( pr2 ( pr1 ( divalgorithmexists ( nattohz n ) ( nattohz m ) p ) ) ) x ) f ( pr2 ( pr2 ( pr2 ( divalgorithmeexists ( nattohz n ) ( nattohz m ) p ) ) ) ) ). exact ( transportf ( fun x : _ => natth ( hzabsval ( pr2 ( pr1 ( divalgorithmexists ( nattohz n ) ( nattohz m ) p ) ) ) ) x ) ( hzabsvalandnattohz m ) ii ). set ( c := y ( hzabsval r ) p' m p ). destruct c as [ ab f ]. destruct ab as [ a b ]. simpl in f. (* split with ( dirprodpair ( ( nattohz n ) - q * ( nattohz m ) ) ( a - b * q ) ) * split with ( dirprodpair b ( a - b * q ) ). assert ( gcd ( nattohz m ) ( nattohz ( hzabsval r ) ) p > ( gcd ( nattohz n ) ( nattohz m ) i ) ) as g. apply isantisymmhzleh. apply ( gcdisgreatest ( nattohz n ) ( nattohz m ) i ). split. apply ( hzdivlinearcombright ( nattohz n ) ( nattohz m ) * ( hzquotientmod ( nattohz m ) p ( nattohz n ) ) r ). exact ( hzdivequationmod ( nattohz m ) p ( nattohz n ) ). apply hzdivand0. apply gcdiscommdiv. unfold r. rewrite ( hzabsvalgeh0 ( hzlehOremaindermod ( nattohz m ) p ( nattohz n ) ) ). apply ( pr2 ( gcdiscommdiv ( nattohz m ) ( hzremaindermod ( nattohz m ) p ( nattohz n ) ) p ( nattohz n ) ) p ). apply gcdiscommdiv. apply gcdisgreatest. split. apply ( pr2 ( gcdiscommdiv ( nattohz m ) * ( hzquotientmod ( nattohz m ) p ( nattohz n ) ) ) ( nattohz ( hzabsval r ) ) ). unfold r. rewrite ( hzabsvalgeh0 ( hzlehOremaindermod ( nattohz m ) p ( nattohz n ) ) ). exact ( hzdivequationmod ( nattohz m ) p ( nattohz n ) ). apply gcdiscommdiv. apply ( hzdivlinearcombright ( nattohz n ) ( nattohz m ) * ( hzquotientmod ( nattohz m ) p ( nattohz n ) ) p ( nattohz n ) ). apply ( pr2 ( gcdiscommdiv ( nattohz m ) * ( hzquotientmod ( nattohz m ) p ( nattohz n ) ) ) ( nattohz ( hzabsval r ) ) ). rewrite <- g. rewrite f. simpl. assert ( nattohz ( hzabsval r ) > ( ( nattohz n ) - ( q * nattohz m ) ) ) as h. rewrite ( hzdivequationmod ( nattohz m ) p ( nattohz n ) ). change ( hzquotientmod ( nattohz m ) p ( nattohz n ) ) with q. change ( hzremaindermod ( nattohz m ) p ( nattohz n ) ) with r. rewrite hzpluscomm. change ( r + nattohz m * q - q * nattohz m ) with ( ( r + nattohz m * q ) + ( - ( q * nattohz m ) ) ). rewrite hzmultcomm. rewrite hzplusassoc. change ( q * nattohz m + - ( q * nattohz m ) ) with ( ( q * nattohz m - ( q * nattohz m ) ) ). rewrite hzminus. rewrite hzplusr0. apply hzabsvalgeh0. apply ( hzlehOremaindermod ( nattohz m ) p ( nattohz n ) ). rewrite h. change ( ( nattohz n - q * nattohz m ) ) with ( ( nattohz n + ( - ( q * nattohz m ) ) ) ). rewrite

```

$m) \dots)$ at 1. rewrite (rngldistr hz). rewrite \leftarrow (hzplusassoc).
 rewrite (hzpluscomm ($a * \text{nattohz } m$)). rewrite
 $\text{rngmultminus}.$ rewrite \leftarrow (hzmattassoc). rewrite \leftarrow
 $\text{rnglmultminus}.$ rewrite (hzplusassoc). rewrite \leftarrow (rngldistr hz).
 change ($b * \text{nattohz } n + (a - b * q) * \text{nattohz } m$) with ($(b * \text{nattohz } n) \% \text{rng} + ((a - (b * q) \% \text{hz}) * \text{nattohz } m) \% \text{rng}$). apply idpath. apply
 goal. Defined.

Lemma $\text{divandhzabsval} (n : \text{hz}) : \text{hzdiv } n (\text{nattohz} (\text{hzabsval } n))$.
 Proof. intros. destruct ($\text{hzlthorgeh } 0 \text{ m}$) as [$\text{left}_m | \text{right}_m$].
 intros $P \ s.$ apply $s.$ split with 1. unfold $\text{hzdiv0}.$ rewrite $\text{hzmultr1}.$
 rewrite $\text{hzabsvalgh0}.$ apply idpath. assumption. intros $P \ s.$ apply
 $s.$ split with ($- 1/\text{hz}$). unfold $\text{hzdiv0}.$ rewrite (rnglmultminus hz).
 rewrite $\text{hzmultr1}.$ rewrite $\text{hzabsvalle0}.$ apply idpath. assumption.
 Defined.

Lemma $\text{bezoutstrong} (m \ n : \text{hz}) (i : \text{hzneq } 0 \ p) : \text{total2} (\text{fun } ab : \text{dirprod hz hz} \Rightarrow (\text{gcd } n \ m \simgt (\text{pr1 } ab) * n + (\text{pr2 } ab) * m))$.
 Proof. intros. assert ($\text{hzneq } 0 (\text{nattohz} (\text{hzabsval } n))$) as $i'.$ intro $f.$ apply $i.$ destruct ($\text{hzneqchoice } 0 \ n \ i$) as [$\text{left}_i | \text{right}_i$]. rewrite hzabsvallh0 in $f.$ rewrite \leftarrow (rngminusminus hz). change 0 with ($- 0$). apply
 $\text{maponpaths}.$ assumption. assumption. rewrite hzabsvalgh0 in $f.$
 $f.$ assumption. assumption. set ($c := (\text{natbezoutstrong} (\text{hzabsval } m) (\text{hzabsval } n \ i'))$). destruct c as [ab]. destruct ab as [$\text{a} \ b$]. simpl in $f.$ assert ($\text{gcd } n \ m \simgt \text{gcd} (\text{nattohz} (\text{hzabsval } n))$ as $\text{nattohz} (\text{hzabsval } m \ i')$) as $g.$ destruct ($\text{hzneqchoice } 0 \ n \ i$) as [$\text{left}_n \ | \ \text{right}_n$]. apply $\text{isantisymmhzh0}.$ apply
 $\text{gcdisgreatest}.$ split. rewrite $\text{hzabsvallh0}.$ apply
 $\text{hzdivandminus}.$ apply $\text{gcdiscommondiv}.$ assumption. destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m \ | \ \text{right}_m$]. rewrite $\text{hzabsvalgh0}.$
 apply ($\text{pr2} (\text{gcdiscommondiv} \ __ \ __)$). assumption. rewrite
 $\text{hzabsvalle0}.$ apply $\text{hzdivandminus}.$ apply ($\text{pr2} (\text{gcdiscommondiv} \ __ \ __)$). assumption. apply $\text{gcdisgreatest}.$ split. apply ($\text{hzdivistrans} \ __ (\text{nattohz} (\text{hzabsval } n)) \ __$). apply $\text{gcdiscommondiv}.$ rewrite
 $\text{hzabsvallh0}.$ rewrite \leftarrow ($\text{rngminusminus hz } n$). apply
 $\text{hzdivandminus}.$ rewrite ($\text{rngminusminus hz } n$). apply $\text{hzdivisrefl}.$
 assumption. apply ($\text{hzdivistrans} \ __ (\text{nattohz} (\text{hzabsval } m)) \ __$). apply ($\text{pr2} (\text{gcdiscommondiv} \ __ \ __)$). destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m \ | \ \text{right}_m$]. rewrite $\text{hzabsvalgh0}.$ apply
 $\text{hzdivisrefl}.$ assumption. apply $\text{isantisymmhzh0}.$ apply
 $\text{gcdisgreatest}.$ split. rewrite $\text{hzabsvallh0}.$ apply
 $\text{gcdiscommondiv}.$ assumption. apply ($\text{hzdivistrans} \ __ (\text{nattohz} (\text{hzabsval } m)) \ __$). destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m \ | \ \text{right}_m$]. rewrite $\text{hzabsvallh0}.$ apply
 $\text{hzdivisrefl}.$ assumption. apply ($\text{hzdivistrans} \ __ (\text{nattohz} (\text{hzabsval } m)) \ __$). apply ($\text{pr2} (\text{gcdiscommondiv} \ __ \ __)$). destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m \ | \ \text{right}_m$]. rewrite $\text{hzabsvalgh0}.$ apply
 $\text{hzdivisrefl}.$ assumption. rewrite $\text{hzabsvalle0}.$ apply $\text{hzdivandminus}.$ apply ($\text{pr2} (\text{gcdiscommondiv} \ __ \ __)$). assumption. apply $\text{hzdivisrefl}.$ apply
 $\text{gcdisgreatest}.$ split. apply ($\text{hzdivistrans} \ __ (\text{nattohz} (\text{hzabsval } n)) \ __$). apply $\text{gcdiscommondiv}.$ rewrite $\text{hzabsvallh0}.$ apply
 $\text{hzdivisrefl}.$ assumption. apply ($\text{hzdivistrans} \ __ (\text{nattohz} (\text{hzabsval } m)) \ __$). apply ($\text{pr2} (\text{gcdiscommondiv} \ __ \ __)$). destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m \ | \ \text{right}_m$]. rewrite $\text{hzabsvalgh0}.$ apply
 $\text{hzdivisrefl}.$ assumption. rewrite $\text{hzabsvalle0}.$ rewrite \leftarrow ($\text{rngminusminus hz } m$). apply $\text{hzdivandminus}.$ rewrite ($\text{rngminusminus hz } m$). apply $\text{hzdivisrefl}.$ assumption. destruct ($\text{hzneqchoice } 0 \ n \ i$) as [$\text{left}_n \ | \ \text{right}_n$]. destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m \ | \ \text{right}_m$].

split with ($\text{dirprodpair} (- a) b$). simpl. assert ($- a * n + b * m \simgt (a * (\text{nattohz} (\text{hzabsval } n)) + b * (\text{nattohz} (\text{hzabsval } m)))$) as 1. rewrite $\text{hzabsvallh0}.$ rewrite $\text{hzabsvalgh0}.$ rewrite (rnglmultminus hz). rewrite \leftarrow (rnglmultminus hz). apply
 idpath. assumption. assumption. rewrite 1. rewrite $g.$ exact
 $f.$ split with ($\text{dirprodpair} (- a) (- b)$). simpl. rewrite 2! (

rnglmultminus hz). rewrite \leftarrow 2! (rnglmultminus hz). rewrite \leftarrow (hzabsvallh0). rewrite \leftarrow (hzabsvalle0). rewrite $g.$ exact
 $f.$ assumption. assumption. destruct ($\text{hzlthorgeh } 0 \ m$) as [$\text{left}_m | \ \text{right}_m$]. split with ($\text{dirprodpair} a \ b$). simpl. rewrite
 $g.$ rewrite $f.$ rewrite 2! $\text{hzabsvalgh0}.$ apply
 idpath. assumption. assumption. split with ($\text{dirprodpair} a (- b)$). rewrite $g.$ rewrite $f.$ simpl. rewrite $\text{hzabsvalgh0}.$ rewrite
 $\text{hzabsvalle0}.$ rewrite (rnglmultminus hz). rewrite \leftarrow (rnglmultminus hz). apply idpath. assumption. assumption. Defined.

(* * * V. Z/nZ *)

Lemma $\text{hzmodisaprop} (p : \text{hz}) (x : \text{hzneq } 0 \ p) (n \ m : \text{hz}) : \text{isaprop} (\text{hzremaindermod } p \ x \ n \simgt (\text{hzremaindermod } p \ x \ m))$. Proof.
 intros. apply $\text{isasethz}.$ Defined.

Definition $\text{hzmod} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{hz} \rightarrow \text{hz} \rightarrow \text{hProp}.$
 Proof. intros $p \ x \ n \ m.$ exact ($\text{hPropair} (\text{hzremaindermod } p \ x \ n \simgt (\text{hzremaindermod } p \ x \ m))$). Defined.

Lemma $\text{hzmodisrefl} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{isrefl} (\text{hzmod } p \ x).$
 Proof. intros. unfold $\text{isrefl}.$ intro $n.$ unfold $\text{hzmod}.$ assert ($\text{hzremaindermod } p \ x \ n \simgt (\text{hzremaindermod } p \ x \ n)$) as $a.$ auto. apply
 $a.$ Defined.

Lemma $\text{hzmodissymm} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{issymm} (\text{hzmod } p \ x).$
 Proof. intros. unfold $\text{issymm}.$ intros $n \ m.$ unfold $\text{hzmod}.$ intro $v.$ assert ($\text{hzremaindermod } p \ x \ m \simgt \text{hzremaindermod } p \ x \ n$) as $a.$ exact ($\text{pathsinv0 } v$). apply $a.$ Defined.

Lemma $\text{hzmodistrans} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{istrans} (\text{hzmod } p \ x).$
 Proof. intros. unfold $\text{istrans}.$ intros $n \ m \ k.$ intros $u \ v.$ unfold $\text{hzmod}.$ unfold hzmod in $u.$ unfold hzmod in $v.$ assert ($\text{hzremaindermod } p \ x \ n \simgt \text{hzremaindermod } p \ x \ k$) as $a.$ exact ($\text{pathscomp0 } u \ v$). apply
 $a.$ Defined.

Lemma $\text{hzmodiseqrel} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{iseqrel} (\text{hzmod } p \ x).$
 Proof. intros. apply $\text{iseqrelconstr}.$ exact ($\text{hzmodistrans } p \ x$). exact ($\text{hzmodisrefl } p \ x$). exact ($\text{hzmodissymm } p \ x$). Defined.

Lemma $\text{hzmodcompatmultl} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{forall } a \ b \ c : \text{hz}, \text{hzmod } p \ x \ a \ b \rightarrow \text{hzmod } p \ x (c * a) (c * b).$ Proof. intros $p \ x \ a \ b \ c \ v.$ unfold $\text{hzmod}.$ change ($\text{hzremaindermod } p \ x (c * a) \simgt \text{hzremaindermod } p \ x (c * b)$). rewrite $\text{hzremaindermodandtimes}.$ rewrite $v.$ rewrite $\leftarrow \text{hzremaindermodandtimes}.$ apply idpath. Defined.

Lemma $\text{hzmodcompatmultr} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{forall } a \ b \ c : \text{hz}, \text{hzmod } p \ x \ a \ b \rightarrow \text{hzmod } p \ x (a * c) (b * c).$ Proof. intros $p \ x \ a \ b \ c \ v.$ rewrite $\text{hzmultcomm}.$ rewrite ($\text{hzmultcomm } b$). apply
 $\text{hzmodcompatmultl}.$ assumption. Defined.

Lemma $\text{hzmodcompatplusl} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{forall } a \ b \ c : \text{hz}, \text{hzmod } p \ x \ a \ b \rightarrow \text{hzmod } p \ x (c + a) (c + b).$ Proof. intros $p \ x \ a \ b \ c \ v.$ unfold $\text{hzmod}.$ change ($\text{hzremaindermod } p \ x (c + a) \simgt \text{hzremaindermod } p \ x (c + b)$). rewrite
 $\text{hzremaindermodandplus}.$ rewrite $v.$ rewrite $\leftarrow \text{hzremaindermodandplus}.$ apply idpath. Defined.

Lemma $\text{hzmodcompatplusr} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{forall } a \ b \ c : \text{hz}, \text{hzmod } p \ x \ a \ b \rightarrow \text{hzmod } p \ x (a + c) (b + c).$ Proof. intros $p \ x \ a \ b \ c \ v.$ rewrite $\text{hzpluscomm}.$ rewrite ($\text{hzpluscomm } b$). apply
 $\text{hzmodcompatplusl}.$ assumption. Defined.

Lemma $\text{hzmodisrngeqrel} (p : \text{hz}) (x : \text{hzneq } 0 \ p) : \text{rngeqrel} (X := \text{hz}).$ Proof. intros. split with ($\text{tpair} (\text{hzmod } p \ x) (\text{hzmodisqrel } p \ x)$). split. split. apply $\text{hzmodcompatplusl}.$ apply

```

hzmodcompatplusr. split. apply hzmodcompatmultl. apply
hzmodcompatmultr. Defined.

Definition hzmodp ( p : hz ) ( x : hzneq 0 p ) := commrngrquot (
hzmodisrngeqrel p x ).

Lemma isdeceqhzmodp ( p : hz ) ( x : hzneq 0 p ) : isdeceq ( hzmodp p x ). Proof. intros. apply ( isdeceqsetquot ( hzmodisrngeqrel p x ) ). intros a b. unfold isdecprop. destruct ( isdeceqhz ( hzremaindermod p x a ) ( hzremaindermod p x b ) ) as [ l | r ]. unfold hzmodisrngeqrel. simpl. split with ( iiii 1 ). intros t. destruct t as [ f | g ]. apply maponpaths. apply isasethz. assert empty. apply g. assumption. contradiction. split with ( iiir r ). intros t. destruct t as [ f | g ]. assert empty. apply r. assumption. contradiction. apply maponpaths. apply isapropneg. Defined.

Definition acommrngr_hzmod ( p : hz ) ( x : hzneq 0 p ) : acommrngr.
Proof. intros. split with ( hzmodp p x ). split with ( tpair _ ( deceqoneqpart ( isdeceqhzmodp p x ) ) ). split. split. intros a b c q. simpl in q. intro f. apply q. rewrite f. apply idpath. intros a b c q. simpl in q. simpl. intro f. apply f. apply idpath. split. intros a b c q. simpl in q. simpl. intros f. apply q. rewrite f. apply idpath. intros a b c q. simpl. simpl in q. intro f. apply q. rewrite f. apply f. Defined.

Lemma hzremaindermodadd ( p : hz ) ( x : hzneq 0 p ) ( a : hz ) ( y : hzdiv p a ) : hzremaindermod p x a ^> 0. Proof. intros. assert ( isaprop ( hzremaindermod p x a ^> 0 ) ) as v. apply isasethz. apply ( y ( hProppair _ v ) ). intro t. destruct t as [ k f ]. unfold hzdiv0 in f. assert ( a ^> ( p * k + 0 ) ) as f'. rewrite f. rewrite hzplus0. apply idpath. set ( e := tpair ( P := ( fun qr : dirprod hz qr => dirprod ( a ^> ( p * pr1 qr + pr2 qr ) ) ( dirprod ( hzleq 0 ( pr2 qr ) ( hzleq ( pr2 qr ) ( nattohz ( hzabsval p ) ) ) ) ) ( dirprodpair k 0 ) ( dirprodpair f' ( dirprodpair ( isreflhzleq 0 ) ( hzabsvalneq 0 p x ) ) ) ) ). assert ( e ^> ( pr1 ( divalgorithm a p x ) ) ) as s. apply ( pr2 ( divalgorithm a p x ) ). set ( w := pathintotalpr1 ( pathsinv0 s ) ). unfold e in w. unfold hzremaindermod. apply ( maponpaths ( fun z : dirprod hz qr => pr2 z w ) ). Defined.

Lemma gcdandprime ( p : hz ) ( x : hzneq 0 p ) ( y : isaprime p ) ( a : hz ) ( q : neg ( hzmod p x a 0 ) ) : gcd p a x ^> 1. Proof. intros. assert ( isaprop ( gcd p a x ^> 1 ) ) as is. apply ( isasethz ). apply ( pr2 y ( gcd p a x ) ( pr1 ( gcdiscommdiv p a x ) ) ( hProppair _ is ) ). intro t. destruct t as [ t0 | t1 ]. apply t0. assert empty. apply q. simpl. assert ( hzremaindermod p x a ^> 0 ) as f. assert ( hzdiv p a ) as u. rewrite <- ( hzdiv p a ) as u. rewrite ( pr2 ( gcdiscommdiv _ _ ) ). rewrite hzremaindermodadd. apply idpath. assumption. rewrite f. rewrite hzgrand0. apply idpath. contradiction. Defined.

Lemma hzremaindermodmultl ( p : hz ) ( x : hzneq 0 p ) ( a b : hz ) : hzremaindermod p ( p * a + b ) ^> hzremaindermod p x b. Proof. intros. assert ( p * a + b ^> ( p * ( a + hzquotientmod p x b ) + hzremaindermod p x b ) ) as f. rewrite hzldistr. rewrite hzplusassoc. rewrite <- ( hzdivequationmod p x b ). apply idpath. rewrite hzremaindermodplus. rewrite hzremaindermodandtimes. rewrite hzgrandselfr. rewrite hzmult0x. rewrite hzgrand0r. rewrite hzplus0l. rewrite hzremaindermoditerated. apply idpath. Defined.

```

```

Lemma hzmodprimeinv ( p : hz ) ( x : hzneq 0 p ) ( y : isaprime p ) ( a : hz ) ( q : neg ( hzmod p x a 0 ) ) : total2 ( fun v : hz => dirprod ( hzmod p x ( a * v 1 ) ( hzmod p x ( v * a ) 1 ) ) ). Proof. intros. split with ( pr2 ( pr1 ( bezoutstrong a p x ) ) ). assert ( 1 ^> ( pr1 ( pr1 ( bezoutstrong a p x ) ) * p + pr2 ( pr1 ( bezoutstrong a p x ) * a ) ) as f'). assert ( 1 ^> gcd p a x ) as f''. apply pathsinv0. apply ( gcdandprime ). assumption. assumption. rewrite f''. apply ( bezoutstrong a p x ). split. rewrite f''. simpl. rewrite ( hzmultcomm ( pr1 ( pri ( bezoutstrong a p x ) ) ) _ ). rewrite ( hzremaindermodandmultl ). rewrite hzmultcomm. apply idpath. rewrite f''. simpl. rewrite hzremaindermodplus. rewrite hzremaindermodandtimes. rewrite hzgrandselfr. rewrite hzmult0x. rewrite hzgrand0r. rewrite hzplus0l. rewrite hzremaindermoditerated. apply idpath. Defined.

Lemma quotientrngrsumdecom ( X : commrngr ) ( R : rngeqrel ( X := X ) ) ( a b : X ) : @op2 ( commrngrquot R ) ( setquotptr R a ) ( setquotptr R b ) ^> ( setquotptr R ( a * b )%rngr ). Proof. intros. auto. Defined.

Definition ahzmod ( p : hz ) ( y : isaprime p ) : afld. Proof. intros. split with ( acommrngr_hzmod p ( isaprimetoneq0 y ) ). split. simpl. intro f. apply ( isirreflhzleq 0 ). assert ( hzleq 0 1 ) as i. apply hzithnsn. change ( 1%rngr ) with ( setquotptr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) 1%hz ) in f. change ( 0%rngr ) with ( setquotptr ( hzmodisrngeqrel ( isaprimetoneq0 y ) ) 0%hz ). assert ( ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) 1%hz 0%hz ) as o. apply ( setquotptrpathsandR ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) 1%hz 0%hz ). assumption. unfold hzmodisrngeqrel in o. simpl in o. assert ( hzremaindermod p ( isaprimetoneq0 y ) 0 ^> 0 ) as o'. rewrite hzgrand0r. apply idpath. rewrite o' in o. assert ( hzremaindermod p ( isaprimetoneq0 y ) 1 ^> 1 ) as o''. assert ( hzleq 1 p ) as v. apply y. rewrite hzgrandr. apply idpath. rewrite o'' in o. assert ( hzleq 0 1 ) as o'''. apply hzithnsn. rewrite o in o'''. assumption. assert ( forall x0 : acommrngr_hzmod p ( isaprimetoneq0 y ), isaprop ( ( x0 # 0)%rngr -> multinvpair ( acommrngr_hzmod p ( isaprimetoneq0 y ) ) x0 ) ) as int. intro a. apply impred. intro q. apply isapropmultinvpair. apply ( setquotunivprop _ ( fun x0 => hProppair _ ( int x0 ) ) ). intro a. simpl. intro q. assert ( neg ( hzmod p ( isaprimetoneq0 y ) a 0 ) ) as q'. intro g. unfold hzmod in g. simpl in g. apply q. change ( 0%rngr ) with ( setquotptr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) 0%hz ). apply ( iscompsetquotpr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) ). apply g. split with ( setquotptr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) ( pr1 ( hzmodprimeinv p ( isaprimetoneq0 y ) y a q' ) ) ). split. simpl. rewrite ( quotientrngrsumdecom hz ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) ). change 1%multmonoid with ( setquotptr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) 1%hz ). apply ( iscompsetquotpr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) ). simpl. change ( pr2 ( pr1 ( bezoutstrong a p ( isaprimetoneq0 y ) ) * a )%hz with ( pr2 ( pr1 ( bezoutstrong a p ( isaprimetoneq0 y ) ) * a )%hz ) ). simpl. rewrite ( quotientrngrsumdecom hz ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) ). change 1%multmonoid with ( setquotptr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) 1%hz ). apply ( iscompsetquotpr ( hzmodisrngeqrel p ( isaprimetoneq0 y ) ) ). change ( a * pr2 ( pr1 ( bezoutstrong a p ( isaprimetoneq0 y ) ) )%rngr with ( a * pr2 ( pr1 ( bezoutstrong a p ( isaprimetoneq0 y ) ) )%hz ). exact ( ( pr1 ( pr2 ( hzmodprimeinv p ( isaprimetoneq0 y ) y a q' ) ) ) ). Defined.

Close Scope hz_scope.
(** END OF FILE *)

```

7.7 The file padics.v

```

(** * padic numbers *)
(** By Alvaro Pelayo, Vladimir Voevodsky and Michael A. Warren *)
(** 2012 *)
(** Settings *)

Add Rec LoadPath "../Generalities". Add Rec LoadPath "../hlevel1".
Add Rec LoadPath "../hlevel2". Add Rec LoadPath
"../Proof_of_Extensionality". Add Rec LoadPath "../Algebra".

Unset Automatic Introduction. (** This line has to be removed for the
file to compile with Coq8.2*)

(** Imports *)
Require Export lemmas.

Require Export fps.

Require Export frac.

Require Export z_mod_p.

(** * I. Several basic lemmas *)
Open Scope hz_scope.

Lemma hzgrandnatsummationOr ( m : hz ) ( x : hzneq 0 m ) ( a : nat -> hz ) ( upper : nat ) : hzremaindermod m x ( natsummation0 upper a ) ~> hzremaindermod m x ( natsummation0 upper ( fun n : nat => hzremaindermod m x ( a n ) ) ). Proof. intros. induction upper. simpl. rewrite hzremaindermoditerated. apply idpath. change ( hzremaindermod m x ( natsummation0 upper a + a ( S upper ) ) ~> hzremaindermod m x ( natsummation0 upper ( fun n : nat => hzremaindermod m x ( a n ) ) + hzremaindermod m x ( a ( S upper ) ) ) ). rewrite hzremaindermodandplus. rewrite IHupper. rewrite <- ( hzremaindermoditerated m x ( a ( S upper ) ) ). rewrite <- hzremaindermodandplus. rewrite hzremaindermoditerated. apply idpath. Defined.

Lemma hzgrandnatsummationOq ( m : hz ) ( x : hzneq 0 m ) ( a : nat -> hz ) ( upper : nat ) : hzquotientmod m x ( natsummation0 upper a ) ~> ( natsummation0 upper ( fun n : nat => hzquotientmod m x ( a n ) ) + hzquotientmod m x ( natsummation0 upper ( fun n : nat => hzremaindermod m x ( a n ) ) ) ). Proof. intros. induction upper. simpl. rewrite <- hzgrandremainderq. rewrite hzplus0. apply idpath. change ( natsummation0 ( S upper ) a ) with ( natsummation0 upper a + a ( S upper ) ). rewrite hzquotientmodandplus. rewrite IHupper. rewrite <- ( hzquotientmod m x ( a n ) ) ( hzquotientmod m x ( a ( S upper ) ) ). rewrite <- ( hzplusassoc ( natsummation0 upper ( fun n : nat => hzquotientmod m x ( a n ) ) ) ( hzquotientmod m x ( a ( S upper ) ) ) ). change ( natsummation0 upper ( fun n : nat => hzquotientmod m x ( a n ) ) + hzquotientmod m x ( a ( S upper ) ) ) with ( natsummation0 ( S upper ) ( fun n : nat => hzquotientmod m x ( a n ) ) ). rewrite hzgrandnatsummationOr. rewrite hzquotientmodandplus. rewrite <- hzgrandremainderq. rewrite hzplus0. rewrite hzremaindermoditerated. rewrite ( hzplusassoc ( natsummation0 ( S upper ) ( fun n : nat => hzquotientmod m x ( a n ) ) ) ( hzquotientmod m x ( natsummation0 upper ( fun n : nat => hzremaindermod m x ( a n ) ) ) ) )

```

(** * II. The carrying operation and induced equivalence relation on formal power series *)

Open Scope rng_scope.

Fixpoint precarry (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz)

$$(n : \text{nat}) : \text{hz} := \text{match } n \text{ with } | 0\% \text{nat} \Rightarrow a | S n \Rightarrow a (S n) + (\text{hzquotientmod } m \text{ is } (\text{precarry } m \text{ is a } n)) \text{ end.}$$

Definition carry (m : hz) (is : hzneq 0 m) : fpcommrnrng hz ->

$$\text{fpcommrnrng } hz := \text{fun } a : \text{fpcommrnrng } hz \Rightarrow \text{fun } n : \text{nat} \Rightarrow$$

$$\text{hzremaindermod } m \text{ is } (\text{precarry } m \text{ is a } n).$$

(* precarry and carry are as described in the following example:

CASE: mod 3

First we normalize the sequence as we go along:

5 6 8 4 (13) 2 2 (remainder 2 mod 3 = 2) 4 1 (remainder 13 mod 3 = 1, quotient 13 mod 3 = 4) 2 2 (remainder 8 mod 3 = 2, quotient 8 mod 3 = 2) 3 1 (remainder 10 mod 3 = 1, quotient 10 mod 3 = 3) 3 0 (remainder 9 mod 3 = 0, quotient 9 mod 3 = 3) 2 2 (remainder 8 mod 3 = 2, quotient 8 mod 3 = 2)

2 2 0 1 2 1 2

Next we first precarry and then carry:

5 6 8 4 (13) 2 2 4 13 2 8 3 (10) 3 9 2 8

48

2 8 9 (10) 8 (13) 2 <-- precarried sequence

2 2 0 1 2 1 2 <-- carried sequence *)

Lemma isapropcarryequiv (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng hz) : isaprop ((carry m is a) ~> (carry m is b)).
Proof. intros. apply (fprop hz). Defined.

Definition carryequiv0 (m : hz) (is : hzneq 0 m) : hrel (fpcommrnrng hz) := fun a b : fpcommrnrng hz => hProppair _ (isapropcarryequiv m is a b).

Lemma carryequiviseqrel (m : hz) (is : hzneq 0 m) : iseql (carryequiv0 m is). Proof. intros. split. split. intros a b c i j. simpl. rewrite i. apply j. intros a. simpl. apply idpath. intros a b i. simpl. rewrite i. apply idpath. Defined.

Lemma carryandremainder (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) (n : nat) : hzremaindermod m is (carry m is a n) ~> carry m is a n. Proof. intros. unfold carry. rewrite hzremaindermoditerated. apply idpath. Defined.

Definition carryequiv (m : hz) (is : hzneq 0 m) : eqrel (fpcommrnrng hz) := eqrelpair _ (carryequiviseqrel m is).

Lemma precarryandcarry (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) : precarry m is (carry m is a) ~> carry m is a. Proof.
intros. assert (forall n : nat, (precarry m is (carry m is a)) n ~> ((carry m is a) n)) as f. intros n. induction n. simpl. apply idpath. simpl. rewrite IHn. unfold carry at 2. rewrite <- hzgrandcarryeq. rewrite hzplusr0. apply idpath. apply (funextfun _ f). Defined.

Lemma hzrandcarryeq (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz)

) (n : nat) : carry m is a n ~> ((m * 0) + carry m is a n).
Proof. intros. rewrite hzmultx0. rewrite hzplusl0. apply idpath.
Defined.

Lemma hzrandcarryineq (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) (n : nat) : dirprod (hzleh 0 (carry m is a n)) (hzltb (carry m is a n) (nattohz (hzabsval m))). Proof.
intros. split. unfold carry. apply (pr2 (pr1 (divalgorithm (precarry m is a n) m is))). unfold carry. apply (pr2 (pr1 (divalgorithm (precarry m is a n) m is))). Defined.

Lemma hzrandcarryq (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) (n : nat) : 0 ~> hzquotientmod m is (carry m is a n). Proof.
intros. apply (hzqrtestq m is (carry m is a n) 0 (carry m is a n)). split. apply hzrandcarryeq. apply hzrandcarryineq. Defined.

Lemma hzrandcarryr (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) (n : nat) : carry m is a ~> hzremaindermod m is (carry m is a n). Proof. intros. apply (hzqrtestr m is (carry m is a n) 0 (carry m is a n)). split. apply hzrandcarryeq. apply hzrandcarryineq. Defined.

Lemma doublecarry (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) : carry m is (carry m is a) ~> carry m is a. Proof. intros. assert (forall n : nat, (carry m is (carry m is a)) n ~> ((carry m is a) n)) as f. intros. induction n. unfold carry. simpl. change (precarry m is (fun n0 : nat => hzremaindermod m is (precarry m is a n0) n)) with ((precarry m is (carry m is a)) n). rewrite precarryandcarry. rewrite <- hzrandcarryq. rewrite hzplusr0. rewrite hzremaindermoditerated. apply idpath. apply (funextfun _ f). Defined.

Lemma carryandcarryequiv (m : hz) (is : hzneq 0 m) (a : fpcommrnrng hz) : carryequiv m is (carry m is a) a. Proof.
intros. simpl. rewrite doublecarry. apply idpath. Defined.

Lemma quotientprecarryplus (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng hz) (n : nat) : hzquotientmod m is (precarry m is (a + b) n) ~> (hzquotientmod m is (precarry m is a) + hzquotientmod m is (precarry m is b n) + hzquotientmod m is (precarry m is (a + carry m is b) n)). Proof. intros. induction n. simpl. change (hzquotientmod m is (a 0%nat + b 0%nat) ~> (hzquotientmod m is (a 0%nat) + hzquotientmod m is (b 0%nat) + hzquotientmod m is (hzremaindermod m is (a 0%nat) + hzremaindermod m is (b 0%nat)))). rewrite hzquotientmodandplus. apply idpath.

change (hzquotientmod m is (a (S n) + b (S n) + hzquotientmod m is (precarry m is (a + b) n)) ~> (hzquotientmod m is (precarry m is (a (S n)) + hzquotientmod m is (precarry m is (a (S n) + carry m is b (S n)) + hzquotientmod m is (precarry m is (carry m is a + carry m is b) n)))). rewrite IHn. rewrite (rngassoc1 hz (a (S n)) (b (S n)) _). rewrite <- (rngassoc1 hz (b (S n))). rewrite (rngcomm1 hz (b (S n)) _). rewrite <- 3! (rngassoc1 hz (a (S n)) _ _). change (a (S n) + hzquotientmod m is (precarry m is a n)) with (precarry m is a (S n)). set (pa := precarry m is a (S n)). rewrite (rngassoc1 hz pa _ (b (S n))). rewrite (rngcomm1 hz _ (b (S n))). change (b (S n) + hzquotientmod m is (precarry m is b n)) with (precarry m is b (S n)). set (pb := precarry m is b (S n)). set (ab := precarry m is (carry m is a + carry m is b)). rewrite (rngassoc1 hz (carry m is a (S n)) (carry m is b (S n)) (hzquotientmod m is (ab n))). rewrite (hzquotientmodandplus m is (carry m is a (S n)) _). unfold carry at 1. rewrite <- hzrandremainderq. rewrite hzplusl0. rewrite (hzquotientmodandplus m is (carry m is b (S n)) _).

```

unfold carry at 1. rewrite <- hzqrandremainderq. rewrite hzplus10.
rewrite ( rngassoc1 bz pa pb _ ). rewrite ( hzquotientmodandplus m is pa _ ). change ( pb + hzquotientmod m is ( ab n ) ) with ( pb + hzquotientmod m is ( ab n ) )%hz. rewrite ( hzquotientmodandplus m is pb ( hzquotientmod m is ( ab n ) ) ). rewrite <- ?! ( rngassoc1 bz ( hzquotientmod m is pa ) _ ). rewrite <- ?! ( rngassoc1 bz ( hzquotientmod m is pa + hzquotientmod m is pb ) _ ). rewrite 2! ( rngassoc1 bz ( hzquotientmod m is pa + hzquotientmod m is pb + hzquotientmod m is ( ab n ) ) _ ). apply ( maponpaths ( fun x : bz => ( hzquotientmod m is pa + hzquotientmod m is pb + hzquotientmod m is ( hzquotientmod m is ( ab n ) ) + x ) ). unfold carry at 1 2. rewrite 2! hzremaindermoditerated. change ( precarry m is b ( S n ) ) with pb. change ( precarry m is a ( S n ) ) with pa. apply ( maponpaths ( fun x : bz => ( hzquotientmod m is ( hzremaindermod m is pb + hzremaindermod m is ( hzquotientmod m is ( ab n ) ))%hz ) + x ) ). apply maponpaths. apply ( maponpaths ( fun x : bz => hzremaindermod m is ( pa + x ) ) ). rewrite ( hzremaindermodandplus m is ( carry m is b ( S n ) ) _ ). unfold carry. rewrite hzremaindermoditerated. rewrite <- ( hzremaindermodandplus m is ( precarry m is b ( S n ) ) _ ). apply idpath. Defined.

```

```

Lemma carryandplus ( m : hz ) ( is : hzneq 0 m ) ( a b : fpsscommrng hz )
: carry m is ( a + b )  $\rightarrow$  carry m is ( carry m is a + carry m is b )
Proof. intros. assert ( forall n : nat, carry m is ( a + b ) n )  $\rightarrow$ 
( carry m is ( carry m is a + carry m is b ) n ) ) as f. intros
n. destruct n. change ( hzremaindermod m is ( a 0%nat + b 0%nat )  $\rightarrow$ 
hzremaindermod m is ( hzremaindermod m is ( a 0%nat + b 0%nat ) + hzremaindermod
m is ( b 0%nat ) ) ). rewrite hzremaindermodandplus. apply idpath.
change ( hzremaindermod m is ( a ( S n ) + b ( S n ) + hzquotientmod m
is ( precarry m is ( a + b ) n ) )  $\rightarrow$  hzremaindermod m is (
hzremaindermod m is ( a ( S n ) + hzquotientmod m is ( precarry m is a
n ) ) + hzremaindermod m is ( b ( S n ) + hzquotientmod m is (
precarry m is b n ) ) + hzquotientmod m is ( precarry m is ( carry m
is a + carry m is b ) n ) ). rewrite quotientprecarryplus. rewrite
(hzremaindermodandplus m is ( hzremaindermod m is ( a ( S n ) +
hzquotientmod m is ( precarry m is a n ) ) + hzremaindermod m is ( b ( S n )
+ hzquotientmod m is ( precarry m is b n ) ) _ ). change
(hzremaindermod m is ( a ( S n ) + hzquotientmod m is ( precarry m is a
n ) ) + hzremaindermod m is ( b ( S n ) + hzquotientmod m is ( precarry m is
b n ) ) with (hzremaindermod m is ( a ( S n ) + hzquotientmod m is
(precarry m is a n))%rng + hzremaindermod m is ( b ( S n ) +
hzquotientmod m is ( precarry m is b n))%rng)%hz. rewrite <-
(hzremaindermodandplus m is ( a ( S n ) + hzquotientmod m is ( precarry m
is a n ) ) ( b ( S n ) + hzquotientmod m is ( precarry m is b n ) ) .
rewrite <- hzremaindermodandplus. change (( a ( S n ) + hzquotientmod
m is ( precarry m is a n))%rng + (b ( S n ) + hzquotientmod m is
(precarry m is b n))%rng + hzquotientmod m is ( precarry m is ( carry m
is a + carry m is b )%rng n ))%hz with (( a ( S n ) + hzquotientmod m is
(precarry m is a n))%rng + (b ( S n ) + hzquotientmod m is ( precarry m
is b n))%rng + hzquotientmod m is ( precarry m is ( carry m is a + carry
m is b )%rng n ))%rng. rewrite <- (rngassoc1 hz ( a ( S n ) +
hzquotientmod m is ( precarry m is a n ) ) (b ( S n ) ) ( hzquotientmod
m is ( precarry m is b n ) ). rewrite (rngassoc1 hz ( a ( S n ) ) ( hzquotientmod
m is ( precarry m is a n ) ) (b ( S n ) ) ). rewrite <- (rngcomm1 hz ( hzquotientmod m is ( precarry m is a n ) ) (b ( S n ) ) ). rewrite <- 3! (rngassoc1 hz). apply idpath. apply (funextfun _ f ). Defined.

```

```
Definition quotientprecarry ( m : hz ) ( is : hzneq 0 m ) ( a :
fpscommrng hz ) : fpscommrng hz := fun x : nat => hzquotientmod m is (
precarry m is a x ).
```

Lemma quotientandtimesrearrangel ($m : \text{hz}$) ($is : \text{hzneq } 0\ m$) ($x\ y : \text{hz}$) : $\text{hzquotientmod}\ m\ is\ (\ x * y\) \sim ((\ \text{hzquotientmod}\ m\ is\ x\) * y + \text{hzquotientmod}\ m\ is\ ((\ \text{hzremaindermod}\ m\ is\ x\) * y))$. Proof.

```

intros. rewrite hzquotientmodandtimes. change (hzquotientmod m is x * hzquotientmod m is y * m + hzremaindermod m is y * hzquotientmod m is x + hzremaindermod m is x * hzquotientmod m is y + hzquotientmod m is (hzremaindermod m is x * hzremaindermod m is y))%hz with
(hzquotientmod m is x * hzquotientmod m is y * m + hzremaindermod m is y * hzquotientmod m is x + hzremaindermod m is x * hzquotientmod m is y + hzquotientmod m is (hzremaindermod m is x * hzremaindermod m is y))%rung. rewrite (rngcomm2_hz (hzremaindermod m is y) (hzquotientmod m is x)). rewrite (rngassocc2_hz). rewrite <- (rgldistr_hz). rewrite (rngcomm2_hz (hzquotientmod m is y) m). change (m * m * hzquotientmod m is y + hzremaindermod m is y)%rung with (m * hzquotientmod m is y + hzremaindermod m is y)%hz. rewrite <- (hzdivequationmod m is y). change (hzremaindermod m is x * y)%rung with (hzremaindermod m is x * y)%hz. rewrite (hzquotientmodandtimes m is (hzremaindermod m is x * y)). rewrite hzremaindermoderated. rewrite <- hzgrandremainderq. rewrite hzmultx0. rewrite 2! hzmult0x. rewrite hzplusl0. rewrite (rngassoc1_hz). change (hzquotientmod m is x * y + (hzremaindermod m is x * hzquotientmod m is y + hzquotientmod m is (hzremaindermod m is x * hzremaindermod m is y)))%hz with (hzquotientmod m is x * y + (hzremaindermod m is x * hzquotientmod m is y + hzquotientmod m is (hzremaindermod m is x * hzremaindermod m is y)))%rung. apply idpath. Defined.

```

```

Lemma natsummationplusshift { R : commr } ( upper : nat ) ( f g :
nat -> R ) : ( natsummation0 ( S upper ) f ) + ( natsummation0 upper g )
-~> ( f 0%nat + ( natsummation0 upper ( fun x : nat : f = f ( S x ) + g x ) ) ). Proof. intros. destruct upper. unfold
natsummation0. simpl. apply ( rngassoc1 R ). rewrite (
natsummationshift0 ( S upper ) f ). rewrite ( rngcomm1 R _ ( f 0%nat ) ). rewrite ( rngassoc1 R ). rewrite natsummationplusdistr. apply
idpath. Defined.

```

```

Lemma precarryandtimesl ( m : hz ) ( is : hzneq 0 m ) ( a b :
fpscommrg hz ) ( n : nat ) : hzquotientmod m is ( precarry m is ( a * b ) n ) >~ ( quotientprecarry m is a * b ) n + hzquotientmod m is ( precarry m is ( carry m is a ) * b ) n ). Proof.
intros. induction n. unfold precarry. change ( ( a * b ) 0%nat ) with
( 0%nat * b 0%nat ). change ( quotientprecarry m is a * b ) 0%nat with
( hzquotientmod m is ( 0%nat ) * b 0%nat ). rewrite
quotientandtimesarrangell. change ( carry m is a * b ) 0%nat with
( zhremaindermod m is ( 0%nat ) * b 0%nat ). apply idpath.

```

```

change ( precarry m is ( a * b ) ( S n ) ) with ( ( a * b ) ( S n )
+ hzquotientmod m is ( precarry m is ( a * b ) n ) ). rewrite
IHn. rewrite <- ( rngassoc1 hz ). assert ( ( ( a * b ) ( S n ) + (
quotientprecarry m is a * b ) n ) ) ~> ( @op2 ( fpscommrung hz ) (
precarry m is a ) b ) ( S n ) ) as f. change ( ( ( a * b ) ( S n ) )
with ( natsummation0 ( S n ) ( fun x : nat => a * x * b ( minus ( S n
) x ) ) ). change ( ( quotientprecarry m is a * b ) n ) with (
natsummation0 n ( fun x : nat => quotientprecarry m is a * x * b (
minus n x ) ) ). rewrite natsummationplusshift. change ( ( @op2 (
fpscommrung hz ) ( precarry m is a ) b ) ( S n ) ) with (
natsummation0 ( S n ) ( fun x : nat => ( precarry m is a ) * x * b (
minus ( S n ) x ) ) ). rewrite natsummationshift0. unfold precarry
at 2. simpl. rewrite <- ( rngcomm1 hz ( a 0%nat * b ( S n ) ) ) _ .
apply ( maponpaths ( fun x : hz => a 0%nat * b ( S n ) + x ) )
apply natsummationpathsupperf. intros k j. unfold
quotientprecarry. rewrite ( rngrdristr hz ). apply idpath. rewrite
f. rewrite hzquotientmodandplus. change ( @op2 ( fpscommrung hz ) (
precarry m is a ) b ) with ( fpstimes hz ( precarry m is a ) b )
. rewrite ( hzquotientandfpstimes1 m is ( precarry m is a ) b )
. change ( @op2 ( fpscommrung hz ) ( carry m is a ) b ) with (
fpstimes hz ( carry is a ) b ) at 1. unfold fpstimes at 1. unfold
carry at 1. change ( fun n0 : nat => let t' := fun m0 : nat => b ( m0
- m0 )%nat in natsummation0 n0 ( fun x : nat => ( hzremaindermod m is

```

```
(precarry m is a x) * t' x)%rng)) with ( carry m is a * b ). change
( (quotientprecarry m is a * b) (S n) ) with ( natsummation0 (S
n) ( fun i : nat => hzquotientmod m is ( precarry m is a i) * b (
S n - i)%nat ) ). rewrite 2! hzplusassoc. apply ( maponpaths ( fun
v : _ => natsummation0 (S n) ( fun i : nat => hzquotientmod m is (
precarry m is a i) * b (S n - i)%nat ) + v ) ). change ( precarry
m is ( carry m is a * b) (S n) ) with ( ( carry m is a * b) (S
n) + hzquotientmod m is ( precarry m is ( carry m is a * b) n ) ). change
((carry m is a * b) (S n) + hzquotientmod m is (precarry m is
(carry m is a * b) n)) with ((carry m is a * b)%rng (S n) +
hzquotientmod m is (precarry m is (carry m is a * b) n)%rng)%hz.
rewrite ( hzquotientmodplus m is ( ( carry m is a * b) (S n) )
(hzquotientmod m is ( precarry m is ( carry m is a * b) n ) ) ). change
( ( carry m is a * b) (S n) ) with ( ( natsummation0 (S n)
( fun i : nat => hzremaindermod m is ( precarry m is a i) * b (S
n - i)%nat ) ). rewrite hzplusassoc. apply ( maponpaths ( fun v : _
=> ( hzquotientmod m is ( natsummation0 (S n) ( fun i : nat =>
hzremaindermod m is ( precarry m is a i) * b (S n - i)%nat ) ) )
+ v ) ). apply ( maponpaths ( fun v : _ => hzquotientmod m is (
hzquotientmod m is ( precarry m is ( carry m is a * b)%rng n ) ) + v
) ). apply maponpaths. apply ( maponpaths ( fun v : _ => v +
hzremaindermod m is ( hzquotientmod m is ( precarry m is ( carry m
is a * b)%rng n ) ) ) ). unfold fpstimes. rewrite
hzgrandnatsummationOr. rewrite ( hzgrandnatsummationOr m is ( fun i
: nat => hzremaindermod m is ( precarry m is a i) * b (S n - i
)%nat ) ). apply maponpaths. apply
natsummationpathsuperfixed. intros j p. change ( hzremaindermod m
is (hzremaindermod m is (precarry m is a j) * b (minus (S n) j))
with (hzremaindermod m is (hzremaindermod m is (precarry m is a
j) * b (S n - j)%nat)%hz). rewrite (hzremaindermodandtimes m is
(hzremaindermod m is (precarry m is a j)) (b (minus (S n) j)
)). rewrite hzremaindermoditerated. rewrite <-
hzremaindermodandtimes. apply idpath. Defined.
```

Lemma carryandtimes1 (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng
hz) : carry m is (a * b) > carry m is (carry m is a * b).

Proof. intros. assert (forall n : nat, carry m is (a * b) n >
carry m is (carry m is a * b) n) as f. intros n. destruct n. unfold
carry at 1 2. change (precarry m is (a * b) 0%nat) with (a 0%nat
* b 0%nat). change (precarry m is (carry m is a * b) 0%nat) with
(carry m is 0%nat * b 0%nat). unfold carry. change (hzremaindermod
m is (precarry m is a 0) * b 0%nat) with (hzremaindermod m is
(precarry m is a 0) * b 0%nat)%hz. rewrite (hzremaindermodandtimes
m is (hzremaindermod m is (precarry m is a 0%nat)) (b 0%nat)).
rewrite hzremaindermoditerated. rewrite <-
hzremaindermodandtimes. apply idpath. Defined.

Lemma carryandtimes1 (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng
hz) : carry m is (a * b) > carry m is (carry m is a * carry m is
b). Proof. intros. rewrite carryandtimes1. rewrite
carryandtimes1. rewrite (rngcomm2 (fpcommrnrng hz)). rewrite
carryandtimes1. rewrite (rngcomm2 (fpcommrnrng hz)). apply
idpath. Defined.

Lemma carryandtimes (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng
hz) : carry m is (a * b) > carry m is (carry m is a * carry m is
b). Proof. intros. rewrite carryandtimes1. rewrite
carryandtimes1. apply idpath. Defined.

Lemma rncarryequiv (m : hz) (is : hzneq 0 m) : @rncarreqrel (
fpcommrnrng hz). Proof. intros. split with (carryequiv m is
). split. split. intros a b c q. simpl. simpl in q. rewrite
carryandplus. rewrite q. rewrite <- carryandplus. apply idpath. intros
a b c q. simpl. rewrite carryandplus. rewrite q. rewrite <-
carryandplus. apply idpath. split. intros a b c q. simpl. rewrite
carryandtimes. rewrite q. rewrite <- carryandtimes. apply
idpath. intros a b c q. simpl. rewrite carryandtimes. rewrite
q. rewrite <- carryandtimes. apply idpath. Defined.

Definition commrnrngofpadicints (p : hz) (is : isaprime p) :=
commrnrngquot (rngcarryequiv p (isaprimetoneq0 is)).

Definition padicplus (p : hz) (is : isaprime p) := @op1 (commrnrngofpadicints p is).

Definition padictimes (p : hz) (is : isaprime p) := @op2 (commrnrngofpadicints p is).

*** III. The apartness relation on p-adic integers *

Definition padicapart0 (p : hz) (is : isaprime p) : hrel (fpcommrnrng hz) := fun a b : _ => (hexists (fun n : nat => (neq _ (carry p (isaprimetoneq0 is) a n) (carry p (isaprimetoneq0 is) b n)))).

Lemma padicapartiscomprel (p : hz) (is : isaprime p) : iscomprel (carryequiv p (isaprimetoneq0 is)) (padicapart0 p is). Proof. intros p is a' b' i j. apply uahp. intro k. apply k. intros u. destruct u as [n u]. apply total2tohexists. split with n. rewrite <- i , <- j. assumption. intro k. apply k. intros u. destruct u as [n u]. apply total2tohexists. split with n. rewrite i, j. assumption. Defined.

Definition padicapart1 (p : hz) (is : isaprime p) : hrel (

```
hzquotientmod m is (precarry m is a j) * b (minus n j)) with (a (S j)
* b (minus n j) + hzquotientmod m is (precarry m is a j) * b (minus
n j))%hz. rewrite <- (hzdistc (a (S j)) (hzquotientmod m is (
precarry m is a j)) (b (minus n j))). rewrite
hzremaindermodandtimes. change (hzremaindermod m is (hzremaindermod m
is (a (S j) + hzquotientmod m is (precarry m is a j)) * hzremaindermod
m is (b (minus n j))))%hz. rewrite <- (hzremaindermod m is (carry m is a (S j) * b
(minus n j))%hz. rewrite <- (hzremaindermoditerated m is (a (S j)
+ hzquotientmod m is (precarry m is a j))). unfold carry. rewrite <-
hzremaindermodandtimes. apply idpath. rewrite
hzremaindermodandplus. rewrite h. rewrite <-
hzremaindermodandplus. unfold carry at 3. rewrite
hzremaindermodandplus m is _ (hzremaindermod m is (precarry m is a
0%nat) * b (minus (S n) 0%nat)). rewrite
hzremaindermodandtimes. rewrite hzremaindermoditerated. rewrite <-
hzremaindermodandtimes. change (precarry m is a 0%nat) with (a
0%nat). rewrite <- hzremaindermodandplus. rewrite hzpluscomm. apply
idpath. rewrite g. rewrite <- hzremaindermodandplus. apply
idpath. apply (funextfun _ _ f). Defined.
```

Lemma carryandtimes (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng
hz) : carry m is (a * b) > carry m is (a * carry m is b).

Proof. intros. rewrite (rngcomm2 (fpcommrnrng hz)). rewrite
carryandtimes1. rewrite (rngcomm2 (fpcommrnrng hz)). apply
idpath. Defined.

Lemma carryandtimes (m : hz) (is : hzneq 0 m) (a b : fpcommrnrng
hz) : carry m is (a * b) > carry m is (carry m is a * carry m is
b). Proof. intros. rewrite carryandtimes1. rewrite
carryandtimes1. apply idpath. Defined.

Lemma rncarryequiv (m : hz) (is : hzneq 0 m) : @rncarreqrel (
fpcommrnrng hz). Proof. intros. split with (carryequiv m is
). split. split. intros a b c q. simpl. simpl in q. rewrite
carryandplus. rewrite q. rewrite <- carryandplus. apply idpath. intros
a b c q. simpl. rewrite carryandplus. rewrite q. rewrite <-
carryandplus. apply idpath. split. intros a b c q. simpl. rewrite
carryandtimes. rewrite q. rewrite <- carryandtimes. apply
idpath. intros a b c q. simpl. rewrite carryandtimes. rewrite
q. rewrite <- carryandtimes. apply idpath. Defined.

Definition commrnrngofpadicints (p : hz) (is : isaprime p) :=
commrnrngquot (rngcarryequiv p (isaprimetoneq0 is)).

Definition padicplus (p : hz) (is : isaprime p) := @op1 (commrnrngofpadicints p is).

Definition padictimes (p : hz) (is : isaprime p) := @op2 (commrnrngofpadicints p is).

*** III. The apartness relation on p-adic integers *

Definition padicapart0 (p : hz) (is : isaprime p) : hrel (fpcommrnrng hz) := fun a b : _ => (hexists (fun n : nat => (neq _ (carry p (isaprimetoneq0 is) a n) (carry p (isaprimetoneq0 is) b n)))).

Lemma padicapartiscomprel (p : hz) (is : isaprime p) : iscomprel (carryequiv p (isaprimetoneq0 is)) (padicapart0 p is). Proof. intros p is a' b' i j. apply uahp. intro k. apply k. intros u. destruct u as [n u]. apply total2tohexists. split with n. rewrite <- i , <- j. assumption. intro k. apply k. intros u. destruct u as [n u]. apply total2tohexists. split with n. rewrite i, j. assumption. Defined.

Definition padicapart1 (p : hz) (is : isaprime p) : hrel (

`commrngofpadicints p is) := quotrel (padicpartiscomprel p is).`

`Lemma isirreflpadicapart0 (p : hz) (is : isaprime p) : isirrefl (padicpart0 p is). Proof. intros. intros a f. simpl in f. assert hfalse as x. apply f. intros u. destruct u as [n u]. apply u. apply idpath. apply x. Defined.`

`Lemma issymmpadicapart0 (p : hz) (is : isaprime p) : issymm (padicpart0 p is). Proof. intros. intros a b f. apply f. intros u. destruct u as [n u]. apply total2toexists. split with n. intros g. apply u. rewrite g. apply idpath. Defined.`

`Lemma iscotranspadicapart0 (p : hz) (is : isaprime p) : iscotrans (padicpart0 p is). Proof. intros. intros a b c f. apply f. intros u. destruct u as [n u]. intros P j. apply j. destruct (isdeceqhz (carry p (isaprimetoneq0 is) a n) (carry p (isaprimetoneq0 is) b n)) as [l | r]. apply ii2. intros Q k. apply k. split with n. intros g. apply u. rewrite l, g. apply idpath. apply iii. intros Q k. apply k. split with n. intros g. apply r. assumption. Defined.`

`Definition padicpart (p : hz) (is : isaprime p) : apart (commrngofpadicints p is). Proof. intros. split with (padicpart1 p is). split. unfold padicpart1. apply (isirreflquotrel (padicpartiscomprel p is) (isirreflpadicapart0 p is)). split. apply (issymmpquotrel (padicpartiscomprel p is) (issymmpadicapart0 p is)). apply (iscotransquotrel (padicpartiscomprel p is) (iscotranspadicapart0 p is)). Defined.`

`Lemma precarryandzero (p : hz) (is : isaprime p) : precarry p (isaprimetoneq0 is) 0 \sim (@rngunel1 (fpscommrung hz)). Proof. intros. assert (forall n : nat, precarry p (isaprimetoneq0 is) 0 n \sim (@rngunel1 (fpscommrung hz) n)) as f. intros n. induction n. unfold precarry. change ((@rngunel1 (fpscommrung hz)) 0%nat) with 0%hz. apply idpath. change ((@rngunel1 (fpscommrung hz) (S n) + hzquotientmod p (isaprimetoneq0 is)) (precarry p (isaprimetoneq0 is))) \sim (@rngunel1 (fpscommrung hz) n) \sim 0%hz. rewrite IHn. change ((@rngunel1 (fpscommrung hz)) n) with 0%hz. rewrite hzrand0q. rewrite hzplusl0. apply idpath. apply (funextfun _ _ f). Defined.`

`Lemma carryandzero (p : hz) (is : isaprime p) : carry p (isaprimetoneq0 is) 0 \sim 0. Proof. intros. unfold carry. rewrite precarryandzero. assert (forall n : nat, (fun n : nat => hzremaindermod p (isaprimetoneq0 is) ((@rngunel1 (fpscommrung hz) n) n) \sim (@rngunel1 (fpscommrung hz) n)) n) as f. intros n. rewrite hzrand0r. unfold carry. change ((@rngunel1 (fpscommrung hz) n)) with 0%hz. apply idpath. apply (funextfun _ _ f). Defined.`

`Lemma precarryandone (p : hz) (is : isaprime p) : precarry p (isaprimetoneq0 is) 1 \sim (@rngunel2 (fpscommrung hz)). Proof. intros. assert (forall n : nat, precarry p (isaprimetoneq0 is) 1 n \sim (@rngunel2 (fpscommrung hz) n)) as f. intros n. induction n. unfold precarry. apply idpath. simpl. rewrite IHn. destruct n. change ((@rngunel2 (fpscommrung hz)) 0%nat) with 1%hz. rewrite hzrand1q. rewrite hzplusr0. apply idpath. change ((@rngunel2 (fpscommrung hz)) (S n)) with 0%hz. rewrite hzrand0q. rewrite hzplusr0. apply idpath. apply (funextfun _ _ f). Defined.`

`Lemma carryandone (p : hz) (is : isaprime p) : carry p (isaprimetoneq0 is) 1 \sim 1. Proof. intros. unfold carry. rewrite precarryandone. assert (forall n : nat, (fun n : nat => hzremaindermod p (isaprimetoneq0 is) ((@rngunel2 (fpscommrung hz) n) n) \sim (@rngunel2 (fpscommrung hz) n)) n) as f. intros n. destruct n. change ((@rngunel2 (fpscommrung hz)) 0%nat) with 1%hz. rewrite hzrand1r. apply idpath. change ((@rngunel2 (`

`fpscommrung hz)) (S n)) with 0%hz. rewrite hzrand0r. apply idpath. apply (funextfun _ _ f). Defined.`

`Lemma padicpartcomputation (p : hz) (is : isaprime p) (a b : fpscommrung hz) : (pr1 (padicpart p is)) (setquotpr (carryequiv p (isaprimetoneq0 is)) a) (setquotpr (carryequiv p (isaprimetoneq0 is)) b) \sim padicpart0 p is a b. Proof. intros. apply uahp. intros i. apply i. intro u. apply u. Defined.`

`Lemma padicpartandplusprecarryl (p : hz) (is : isaprime p) (a b c : fpscommrung hz) (n : nat) (x : neq _ (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is)) a + carry p (isaprimetoneq0 is) b) n) ((precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is)) a + carry p (isaprimetoneq0 is) c)) n) : (padicpart0 p is) b c. Proof. intros. set (P := fun x : nat => neq hz (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is)) a + carry p (isaprimetoneq0 is) b) x) (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is) a + carry p (isaprimetoneq0 is) c) x). assert (isdecatprop P) as isdec. intros m. destruct (isdeceqhz (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is)) a + carry p (isaprimetoneq0 is) b) m) (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is) + carry p (isaprimetoneq0 is) c) m) as [l | r]. apply ii2. intros j. apply j. assumption. apply iii. assumption. set (leexists := leastelementprinciple n P isdec x). apply leexists. intro k. destruct k as [k' k'']. destruct k' as [k' k'']. destruct k. apply total2toexists. split with 0%nat. intros i. apply k'. change (carry p (isaprimetoneq0 is)) a 0%nat + carry p (isaprimetoneq0 is) b 0%nat \sim (carry p (isaprimetoneq0 is) a 0%nat + carry p (isaprimetoneq0 is) c 0%nat). rewrite i. apply idpath. apply total2toexists. split with (S k). intro i. apply (k' k). apply natlthns. intro j. apply k'. change (carry p (isaprimetoneq0 is)) (S k) + hzquotientmod p (isaprimetoneq0 is) (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is) a + carry p (isaprimetoneq0 is) k) \sim (carry p (isaprimetoneq0 is) a (S k) + carry p (isaprimetoneq0 is) c (S k) + hzquotientmod p (isaprimetoneq0 is) (precarry p (isaprimetoneq0 is) (carry p (isaprimetoneq0 is) a + carry p (isaprimetoneq0 is) c) k)). rewrite i. rewrite j. apply idpath. Defined.`

`Lemma padicpartandplusprecarryr (p : hz) (is : isaprime p) (a b c : fpscommrung hz) (n : nat) (x : neq _ (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is)) b + carry p (isaprimetoneq0 is)) (precarry p (isaprimetoneq0 is)) (carry p (isaprimetoneq0 is)) c + carry p (isaprimetoneq0 is) a)) n) : (padicpart0 p is) b c. Proof. intros. rewrite 2! (rngcommi (fpscommrung hz) _ (carry p (isaprimetoneq0 is) a)) in x. apply (padicpartandplusprecarryl p is a b c n x). Defined.`

`Lemma commrngquotprandop1 { A : commrng } (R : @rngeqrel A) (a b : A) : (@op1 (commrngquot R)) (setquotpr (pr1 R) a) (setquotpr (pr1 R) b) \sim setquotpr (pr1 R) (a * b). Proof. intros. change (@op1 (commrngquot R)) with (setquotfun2 R R (@op1 A) (pr1 (iscomp2binoptransrel (pr1 R) (eqreltrans _) (pr2 R)))). unfold setquotfun2. rewrite setquotuniv2comm. apply idpath. Defined.`

`Lemma commrngquotprandop2 { A : commrng } (R : @rngeqrel A) (a b : A) : (@op2 (commrngquot R)) (setquotpr (pr1 R) a) (setquotpr (pr1 R) b) \sim setquotpr (pr1 R) (a * b). Proof. intros. change (@op2 (commrngquot R)) with (setquotfun2 R R (@op2 A) (pr2 (iscomp2binoptransrel (pr1 R) (eqreltrans _) (pr2 R)))). unfold setquotfun2. rewrite setquotuniv2comm. apply idpath. Defined.`

```

Lemma setquotprandpadicplus ( p : hz ) ( is : isaprime p ) ( a b :
fpscommrng hz ) : (@op1 ( commrngofadicints p is )) ( setquotpr
carryequiv p ( isaprimetoneq0 is ) ) a ( setquotpr ( carryequiv p (
isaprimetoneq0 is ) ) b ) > setquotpr ( carryequiv p ( isaprimetoneq0
is ) ) ( a + b ). Proof. intros. apply commrngquotprandop1.
Defined.

```

```

Lemma setquotprandpadictimes ( p : hz ) ( is : isaprime p ) ( a b :
fpscommrng hz ) : (@op2 ( commrngofadicints p is )) ( setquotpr
carryequiv p ( isaprimetoneq0 is ) ) a ( setquotpr ( carryequiv p (
isaprimetoneq0 is ) ) b ) > setquotpr ( carryequiv p ( isaprimetoneq0
is ) ) ( a * b ). Proof. intros. apply commrngquotprandop2.
Defined.

```

```

Lemma padicplusbinopapart0 ( p : hz ) ( is : isaprime p ) ( a b c :
fpscommrng hz ) ( u : padicpart0 p is ( a + b ) ( a + c ) ) :
padicpart0 p is b c. Proof. intros. apply u. intros n. destruct n
as [ n n' ]. set ( P := fun x : nat => neq hz ( carry p (
isaprimetoneq0 is ) ( a + b ) x ) ( carry p ( isaprimetoneq0 is ) ( a +
c ) x ) ). assert ( isdecnatprop P ) as isdec. intros m. destruct
isdeceqhz ( carry p ( isaprimetoneq0 is ) ( a + b ) m ) ( carry p (
isaprimetoneq0 is ) ( a + c ) m ) as [ l | r ]. apply ii2. intros
j. apply j. assumption. apply iii. assumption.

```

```

set ( le := leastelementprinciple n P isdec n'). apply le. intro
k. destruct k as [ k k' ]. destruct k' as [ k' k'' ]. destruct k.
apply total2toexists. split with 0%nat. intros j. apply k'. unfold
carry. unfold precarry. change ( ( a + b ) 0%nat ) with ( a 0%nat
+ b 0%nat ). change ( ( a + c ) 0%nat ) with ( a 0%nat + c 0%nat
). unfold carry in j. unfold precarry in j. rewrite
hzremaindermodandplus. rewrite j. rewrite <-
hzremaindermodandplus. apply idpath.

```

```

destruct ( isdeceqhz ( carry p ( isaprimetoneq0 is ) b ( S k ) ) )
(carry p ( isaprimetoneq0 is ) c ( S k ) ) as [ l | r ]. apply (
padicpartandplusprecarryl p is a b c k ). intros j. apply
k'. rewrite ( carryandplus ). unfold carry at 1. change (
hzremaindermod p ( isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is
) a ( S k ) + carry p ( isaprimetoneq0 is ) b ( S k ) +
hzquotientmod p ( isaprimetoneq0 is ) ( precarry p ( isaprimetoneq0
is ) ( carry p ( isaprimetoneq0 is ) a + carry p ( isaprimetoneq0 is
) b ) k ) ) > carry p ( isaprimetoneq0 is ) ( a + c ) ( S k ) .
rewrite 1. rewrite j. rewrite ( carryandplus p ( isaprimetoneq0 is
) a c ). unfold carry at 5. change ( precarry p ( isaprimetoneq0 is )
(carry p ( isaprimetoneq0 is ) a + carry p ( isaprimetoneq0 is ) c
) ( S k ) ) with ( carry p ( isaprimetoneq0 is ) a ( S k ) + carry p
( isaprimetoneq0 is ) c ( S k ) + hzquotientmod p ( isaprimetoneq0 is
) ( precarry p ( isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is
) a + carry p ( isaprimetoneq0 is ) c ) k ) ). apply idpath.
apply total2toexists. split with ( S k ). assumption. Defined.

```

```

Lemma padicplusbinopapart1 ( p : hz ) ( is : isaprime p ) :
isbinopapart1 ( padicpart p is ) ( padicplus p is ). Proof.
intros. unfold isbinopapart1. assert ( forall x x' x'':
commrngofadicints p is, isaprop ( ( pr1 ( padicpart p is ) ) (
padicplus p is x x' ) ( padicplus p is x x'') -> ( ( pr1 ( padicpart
p is ) ) x' x'' ) ) ) as int. intros. apply impred. intros. apply (
pr1 ( padicpart p is ) ). apply ( setquotuniv3prop _ ( fun x x' x''
=> hProppair _ ( int x x' x'' ) ) ). intros a b c. change ( pr1
(padicpart p is) ( padicplus p is ) ( setquotpr ( rngcarryequiv p
(isaprimetoneq0 is) ) a ) ( setquotpr ( rngcarryequiv p ( isaprimetoneq0
is) ) b ) ( padicplus p is ) ( setquotpr ( rngcarryequiv p ( isaprimetoneq0
is) ) a ) ( setquotpr ( rngcarryequiv p ( isaprimetoneq0 is) ) c ) ) -> pr1
(padicpart p is) ( setquotpr ( rngcarryequiv p ( isaprimetoneq0 is) ) b )
( setquotpr ( rngcarryequiv p ( isaprimetoneq0 is) ) c ) ). unfold

```

```

padicplus. rewrite 2! setquotprandpadicplus. rewrite 2!
padicpartcomputation. apply padicplusisbinopapart0. Defined.

```

```

Lemma padicplusbinopapartr ( p : hz ) ( is : isaprime p ) :
isbinopapartr ( padicpart p is ) ( padicplus p is ). Proof.
intros. unfold isbinopapartr. intros a b c. unfold padicplus. rewrite
(@rngcomm1 ( commrngofadicints p is ) b a ). rewrite (@rngcomm1 (
commrngofadicints p is ) c a ). apply padicplusisbinopapart1.
Defined.

```

```

Lemma padicpartandtimesprecarryl ( p : hz ) ( is : isaprime p ) ( a b
c : fpscommrng hz ) ( n : nat ) ( x : neq _ ( precarry p (
isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is ) a * carry p (
isaprimetoneq0 is ) b ) n ) ( precarry p ( isaprimetoneq0 is ) ( carry p
(isaprimetoneq0 is ) b ) n ) ) ( precarry p ( isaprimetoneq0 is ) c ) ) :
padicpart0 p is b c. Proof. intros. set ( P := fun x : nat => neq hz
( precarry p ( isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is ) a *
carry p ( isaprimetoneq0 is ) b ) x ) ( precarry p ( isaprimetoneq0
is ) ( carry p ( isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) c ) x
)). assert ( isdecnatprop P ) as isdec. intros m. destruct ( isdeceqhz
( precarry p ( isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is ) a * carry
p ( isaprimetoneq0 is ) b ) m ) ( precarry p ( isaprimetoneq0 is ) ( carry p
( isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) c ) m ) ) as [ l | r
]. apply ii2. intros j. apply j. assumption. apply iii. assumption.
set ( leexists := leastelementprinciple P isdec x ). apply
leexists. intro k. destruct k as [ k k' ]. destruct k' as [ k' k'' ]
]. induction k. apply total2toexists. split with 0%nat. intros i.
apply k'. change ( carry p ( isaprimetoneq0 is ) a 0%nat * carry p
( isaprimetoneq0 is ) b 0%nat ) > ( carry p ( isaprimetoneq0 is ) a 0%nat *
carry p ( isaprimetoneq0 is ) c 0%nat ). rewrite i. apply idpath. set
( Q := ( fun o : nat => hProppair ( carry p ( isaprimetoneq0 is ) b o
-> carry p ( isaprimetoneq0 is ) c o ) ( isaseth2 _ _ ) ) ). assert
( isdecnatprop Q ) as isdec'. intro o. destruct ( isdeceqhz ( carry p
( isaprimetoneq0 is ) b o ) ( carry p ( isaprimetoneq0 is ) c o ) ) as [
l | r ]. apply iiii. assumption. apply ii2. assumption. destruct
isdecbdqddec ( S k ) as [ l | r ]. assert hfalse as xx. apply
( k' k ). apply natlthsn. intro j. apply k'. change ( ( natsummation0
( S k ) ( fun x : nat => carry p ( isaprimetoneq0 is ) a x * carry p
( isaprimetoneq0 is ) b ( minus ( S k ) x ) ) ) + hzquotientmod p
( isaprimetoneq0 is ) ( precarry p ( isaprimetoneq0 is ) ( carry p
( isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) b k ) ) >
(( natsummation0 ( S k ) ( fun x : nat => carry p ( isaprimetoneq0
is ) a x * carry p ( isaprimetoneq0 is ) c ( minus ( S k ) x ) ) ) +
hzquotientmod p ( isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is ) a * carry p
( isaprimetoneq0 is ) c k ) ) ). assert ( natsummation0 ( S k ) ( fun
x0 : nat => carry p ( isaprimetoneq0 is ) a x0 * carry p ( isaprimetoneq0
is ) b ( minus ( S k ) x0 ) ) > natsummation0 ( S k ) ( fun x0 : nat =>
carry p ( isaprimetoneq0 is ) a x0 * carry p ( isaprimetoneq0 is ) c
( minus ( S k ) x0 ) ) ) as f. apply natsummationpathsuperfixed. intros
y. rewrite ( l ( minus ( S k ) m ) ). apply idpath. apply
minusleh. rewrite f. rewrite j. apply idpath. contradiction.

```

```

apply r. intros o. destruct o as [ o o' ]. apply
total2toexists. split with o. apply o'. Defined.

```

```

Lemma padictimesbinopapart0 ( p : hz ) ( is : isaprime p ) ( a b c :
fpscommrng hz ) ( u : padicpart0 p is ( a * b ) ( a * c ) ) :
padicpart0 p is b c. Proof. intros. apply u. intros n. destruct n
as [ n n' ]. destruct n. apply total2toexists. split with
0%nat. intros j. apply n'. rewrite carryandtimes. rewrite (
carryandtimes p ( isaprimetoneq0 is ) a c ). change ( hzremaindermod p
( isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is ) a 0%nat * carry
p ( isaprimetoneq0 is ) b 0%nat ) > hzremaindermod p ( isaprimetoneq0
is ) ( carry p ( isaprimetoneq0 is ) a 0%nat * carry p ( isaprimetoneq0
is ) c 0%nat ) ). rewrite j. apply idpath. set ( Q :=

```

```

        ( fun o : nat => hProppair ( carry p ( isaprimetoneq0 is ) b o ~>
            carry p ( isaprimetoneq0 is ) c o ) ( isasethz _ _ ) ). assert (
            isdecnatprop Q ) as isdec'. intro o. destruct ( isdeceqhz ( carry p (
            isaprimetoneq0 is ) b o ) ( carry p ( isaprimetoneq0 is ) c o ) ) as [ l | r ]. apply iii. assumption. apply ii2. assumption. destruct (
            isdecisbndqdec Q isdec' ( S n ) ) as [ l | r ]. apply (
            padicpartandtimesprecarryl p is a b c n ). intros j. assert hffalse as
            xx. apply n'. rewrite carryandtimes. rewrite ( carryandtimes p (
            isaprimetoneq0 is ) a c ). change ( hzremaindermod p ( isaprimetoneq0
            is ) ( natsummation0 ( S n ) ( fun x : nat => carry p ( isaprimetoneq0
            is ) a x * carry p ( isaprimetoneq0 is ) b ( minus ( S n ) x ) ) +
            hzquotientmod p ( isaprimetoneq0 is ) ( precarry p ( isaprimetoneq0 is )
            ( carry p ( isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) b
            ) n ) ) ~> ( hzremaindermod p ( isaprimetoneq0 is ) ( natsummation0
            ( S n ) ( fun x : nat => carry p ( isaprimetoneq0 is ) a x * carry p (
            isaprimetoneq0 is ) c ( minus ( S n ) x ) ) + hzquotientmod p (
            isaprimetoneq0 is ) ( precarry p ( isaprimetoneq0 is ) ( carry p (
            isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) c ) n ) ) ). rewrite
            j. assert ( natsummation0 ( S n ) ( fun x0 : nat => carry p (
            isaprimetoneq0 is ) a x0 * carry p ( isaprimetoneq0 is ) b ( minus ( S n
            ) x0) ) ~> natsummation0 ( S n ) ( fun x0 : nat => carry p (
            isaprimetoneq0 is ) a x0 * carry p ( isaprimetoneq0 is ) c ( minus ( S n
            ) x0) ) ) as f. apply natsummationpathsupperfixed. intros m y. rewrite
            ( 1 ( minus ( S n ) m ) ). apply idpath. apply minusleh. rewrite
            f. apply idpath. contradiction. apply r. intros k. destruct k as [ k
            k' ]. apply total2toexists. split with k. apply k'. Defined.

Lemma padictimesisbinopaprtl ( p : hz ) ( is : isaprime p ) :
isbinopaprtl ( padicpart p is ) ( padictimes p is ). Proof.
intros. unfold isbinopaprtl. assert ( forall x x' x' :
commrngofpadicints p is, isaprop ( ( pri ( padicpart p is ) ) (
padictimes p is x x' ) ( padictimes p is x x' ) ) ~> ( ( pr1 (
padicpart p is ) x x' ) ) ) as int. intros. apply
impred. intros. apply ( pr1 ( padicpart p is ) ). apply (
setquotuniv3prop _ ( fun x x' x' ~> hProppair _ ( int x x' x' ) ) ). intros a b c. change ( pr1 ( padicpart p is ) (padictimes p is
(setquotpr (carryequiv p (isaprimetoneq0 is)) a) (setquotpr
(carryequiv p (isaprimetoneq0 is)) b)) (padictimes p is (setquotpr
(carryequiv p (isaprimetoneq0 is)) a) (setquotpr (carryequiv p
(isaprimetoneq0 is)) a) (setquotpr (carryequiv p
(isaprimetoneq0 is)) b) (setquotpr (carryequiv p
(isaprimetoneq0 is)) c)) ~> pr1 (padicpart p is) (setquotpr
(carryequiv p (isaprimetoneq0 is)) b) (setquotpr (carryequiv p
(isaprimetoneq0 is)) c)). unfold padictimes. rewrite ?!
setquotprandpadictimes. rewrite 2! padicpartcomputation. intros j.
apply ( padictimesisbinopaprt0 p is a b c j ). Defined.

Lemma padictimesisbinopaptr ( p : hz ) ( is : isaprime p ) :
isbinopaptr ( padicpart p is ) ( padictimes p is ). Proof.
intros. unfold isbinopaptr. intros a b c. unfold padictimes. rewrite
( @rngcomm2 ( commrngofpadicints p is ) b a ). rewrite ( @rngcomm2 (
commrngofpadicints p is ) c a ). apply padictimesisbinopaprtl.
Defined.

Definition accomrrngofpadicints ( p : hz ) ( is : isaprime p ) :
acomrrng. Proof. intros. split with ( commrngofpadicints p is
). split with ( padicpart p is ). split. split. apply (
padicplusisbinopaprtl p is ). apply ( padicplusisbinopaptr p is ). split.
apply ( padictimesisbinopaprtl p is ). apply (
padictimesisbinopaptr p is ). Defined.

(** * IV. The apartness domain of p-adic integers and the Heyting
field of p-adic numbers *)

Lemma precarryandzeromultl ( p : hz ) ( is : isaprime p ) ( a b :
fpscommrrng hz ) ( n : nat ) ( x : forall m : nat, natlth m n ->
( carry p ( isaprimetoneq0 is ) a m ~> 0%hz ) ) : forall m : nat, natlth
m n -> precarry p ( isaprimetoneq0 is ) ( fpstimes hz ( carry p
isaprimetoneq0 is ) a ) ( carry p ( isaprimetoneq0 is ) b ) ) m ~>
0%hz. Proof. intros p is a b n x m y. induction m. simpl. unfold
fpstimes. simpl. rewrite ( x 0%nat y ). rewrite hzmult0x. apply
idpath. change ( natsummation0 ( S m ) ( fun z : nat => ( carry p (
isaprimetoneq0 is ) a z ) * ( carry p ( isaprimetoneq0 is ) b ( minus
( S m ) z ) ) ) + hzquotientmod p ( isaprimetoneq0 is ) ( precarry p (
isaprimetoneq0 is ) ( fpstimes hz ( carry p ( isaprimetoneq0 is ) a )
( carry p ( isaprimetoneq0 is ) b ) ) m ) ) ~> 0%hz ). assert ( natlth
m n ) as u. apply ( istransnatith _ ( S m ) _ ). apply
natlthsn. assumption. rewrite ( IHm u ). rewrite hzqrand0q. rewrite
hzplus0. assert ( natsummation0 ( S m ) ( fun z : nat => carry p
(isaprimetoneq0 is ) a z * carry p ( isaprimetoneq0 is ) b ( minus ( S m
) z ) ) ) ~> ( natsummation0 ( S m ) ( fun z : nat => 0%hz ) ) ) as f.
apply natsummationpathsupperfixed. intros k v. assert ( natlth k n )
as uu. apply ( natlehlthtrans _ ( S m ) _ ). assumption. assumption. rewrite
( x k uu ). rewrite hzmult0x. apply idpath. rewrite f. rewrite
natsummationae0bottom. apply idpath. intros k l. apply idpath. Defined.

Lemma precarryandzeromultl ( p : hz ) ( is : isaprime p ) ( a b :
fpscommrrng hz ) ( n : nat ) ( x : forall m : nat, natlth m n ->
( carry p ( isaprimetoneq0 is ) b m ~> 0%hz ) ) : forall m : nat, natlth
m n -> precarry p ( isaprimetoneq0 is ) ( fpstimes hz ( carry p (
isaprimetoneq0 is ) a ) ( carry p ( isaprimetoneq0 is ) b ) ) m ~>
0%hz. Proof. intros p is a b n x m y. induction m. simpl. change ( fpstimes hz (carry p
(isaprimetoneq0 is) a) (carry p (isaprimetoneq0 is) b) ) with ( (carry
p (isaprimetoneq0 is) a) * (carry p (isaprimetoneq0 is) b)). rewrite
( @rngcomm2 (fpscommrrng hz) ) (carry p (isaprimetoneq0 is) a) (carry
p (isaprimetoneq0 is) b)). apply ( precarryandzeromultl p is
b a n x m y). Defined.

Lemma hzfpstimesnonzero ( a : fpscommrrng hz ) ( k : nat ) ( is :
dirprod ( neq hz ( a k ) 0%hz ) ( forall m : nat, natlth m k -> ( a m
) ~> 0%hz ) ) : forall k' : nat, forall b : fpscommrrng hz , forall is' :
dirprod ( neq hz ( b k' ) 0%hz ) ( forall m : nat, natlth m k' -> ( b m
) ~> 0%hz ), ( a * b ) ( k + k' )%nat ~> ( a k ) * ( b k' ). Proof.
intros a b is. induction k. induction k'. intros. destruct
k. simpl. apply idpath. rewrite natplus0. change ( natsummation0 k (
fun x : nat => a x * b ( minus ( S k ) x ) ) + a ( S k ) * b ( minus (
S k ) ( S k ) ) ) ~> a ( S k ) * b 0%nat. assert ( natsummation0 k (
fun x : nat => a x * b ( minus ( S k ) x ) ) ~> natsummation0 k ( fun
x : nat => 0%hz ) ) as i0. apply natsummationpathsupperfixed. intros m
i. assert ( natlth m ( S k ) ) as i0. apply ( natlehlthtrans _ k -
). assumption. apply natlthsn. rewrite ( (pr2 is) m i0 ). rewrite
hzmult0x. apply idpath. rewrite f. rewrite
natsummationae0bottom. rewrite hzplus0. rewrite minusnn0. apply
idpath. intros m i. apply idpath. intros. rewrite natplusnn. change
( natsummation0 ( k + k' )%nat ( fun x : nat => a x * b ( minus ( S k
+ k' ) x ) ) + a ( S k + k' )%nat * b ( minus ( S k + k' ) ( S k + k'
) ) ~> a k * b ( S k' ) ). set ( b' := fpsshift b ). rewrite
minusnn0. rewrite ( (pr2 is') 0%nat ( natlehlthtrans 0 k' ( S k' ) (
natlehn0 k' ) ( natlthsn k' ) ) ). rewrite hzmultx0. rewrite
hzplus0. assert ( natsummation0 ( k + k' )%nat ( fun x : nat => a x
* b ( minus ( S k + k' ) ) ) ~> fpstimes hz a b' ( k + k' )%nat ) as f.
apply natsummationpathsupperfixed. intros m v. change ( S k + k'
)%nat with ( S ( k + k' ) ). rewrite <( pathssminus ( k + k' )%nat m
). apply idpath. apply ( natlehlthtrans _ ( k + k' )%nat -
). assumption. apply natlthsn. rewrite f. apply ( IHk' b'
). split. apply is'. intros m v. unfold b'. unfold fpsshift. apply
is'. assumption. Defined.

Lemma hzfpstimeswhenzero ( a : fpscommrrng hz ) ( m k : nat ) ( is :
forall m : nat, natlth m k -> ( a m ) ~> 0%hz ) : forall b :
fpscommrrng hz, forall k' : nat, forall is' : ( forall m : nat, natlth
m k' -> ( b m ) ~> 0%hz ) , natlth m ( k + k' )%nat -> ( a * b ) m ~>
0%hz. Proof. intros a m. induction m. intros k. intros is b k' is'

```

```
j. change ( a 0%nat * b 0%nat ~> 0%hz ). destruct k. rewrite ( is' 0%nat j ). rewrite hzmultx0. apply idpath. assert ( natlth 0 ( S k ) ) as i. apply ( natlehltrans _ k _ ). apply natlehOn. apply natlthnsn. rewrite ( is 0%nat i ). rewrite hzmult0x. apply idpath.
```

```
intros k is b' is' j. change ( natsummation0 ( S m ) ( fun x : nat => a x * b ( minus ( S m ) x ) ) ~> 0%hz ). change ( natsummation0 m ( fun x : nat => a x * b ( minus ( S m ) x ) ) + a ( S m ) * b ( minus ( S m ) ( S m ) ) ~> 0%hz ) as g. assert ( a ( S m ) * b ( minus ( S m ) ( S m ) ) ~> 0%hz ) as g. destruct k. destruct k'. assert empty. apply ( negnatgthOn ( S m ) j ). contradiction. rewrite minusmn0. rewrite ( is' 0%nat ( natlehltrans 0%nat k' ( S k' ) ( natlehOn k' ) ) ). rewrite hzmultx0. apply idpath. destruct k'. rewrite natplusr0 in j. rewrite ( is ( S m ) j ). rewrite hzmult0x. apply idpath. rewrite minusmn0. rewrite ( is' 0%nat ( natlehltrans 0%nat k' ( S k' ) ( natlehOn k' ) ( natlthnsn k' ) ) ). rewrite hzmultx0. apply idpath. rewrite g. rewrite hzplusr0. set ( b' := fpshift b ). assert ( natsummation0 m ( fun x : nat => a x * b ( minus ( S m ) x ) ) ~> natsummation0 m ( fun x : nat => a x * b' ( minus ( S m ) x ) ) ) as f. apply natsummationpathsupperfixed. intros n i. unfold b'. unfold fpshift. rewrite pathssminusn. apply idpath. apply ( natlehltrans _ m _ ). assumption. apply natlthnsn. rewrite f. change ( ( a * b' ) m ~> 0%hz ). assert ( natlth m ( k + k' ) ) as one. apply istransnath_ ( S m ) _ . apply natlthnsn. assumption. destruct k'. assert ( forall m : nat, natlth m 0%nat ~> b' m ~> 0%hz ) as two. intros m0 j0. assert empty. apply ( negnatgthOn m0 ). assumption. contradiction. apply ( IHm k is b' 0%nat two one ). assert ( forall m : nat, natlth m k' ~> b' m ~> 0%hz ) as three. intros m0 j0. change ( b ( S m0 ) ~> 0%hz ). apply is'. assumption. assert ( natlth m ( k + k' ) 0%nat ) as three. rewrite natplusnsn in j. apply j. apply ( IHm k is b' k' two three ). Defined.
```

```
Lemma precarryandzeromult ( p : hz ) ( is : isaprime p ) ( a b : fpcommrnrng hz ) ( k k' : nat ) ( x : forall m : nat, natlth m k ~> carry p ( isaprimetoneq0 is ) a m ~> 0%hz ) ( x' : forall m : nat, natlth m k' ~> carry p ( isaprimetoneq0 is ) b m ~> 0%hz ) : forall m : nat, natlth m ( k + k' ) 0%nat ~> precarry p ( isaprimetoneq0 is ) ( fpstimes hz ( carry p ( isaprimetoneq0 is ) a ) ( carry p ( isaprimetoneq0 is ) b ) ) m ~> 0%hz. Proof. intro p is a b k k' x x' m i. induction m. apply ( hzfpstimeswhenzero ( carry p ( isaprimetoneq0 is ) a ) 0%nat k x ( carry p ( isaprimetoneq0 is ) b ) k' x' i ). change ( ( carry p ( isaprimetoneq0 is ) a ) * ( carry p ( isaprimetoneq0 is ) b ) ) ( S m ) + hzquotientmod p ( isaprimetoneq0 is ) ( precarry p ( isaprimetoneq0 is ) ( fpstimes hz ( carry p ( isaprimetoneq0 is ) a ) ( carry p ( isaprimetoneq0 is ) b ) ) m ) ~> 0%hz. rewrite ( hzfpstimeswhenzero ( carry p ( isaprimetoneq0 is ) a ) ( S m ) k x ( carry p ( isaprimetoneq0 is ) b ) k' x' i ). rewrite hzplusl0. assert ( natlth m ( k + k' ) 0%nat ) as one. apply istransnath_ ( S m ) _ . apply natlthnsn. assumption. rewrite ( IHm one ). rewrite hzgrand0q. apply idpath. Defined.
```

```
Lemma primedivorcprime ( p a : hz ) ( is : isaprime p ) : hdisj ( hzdiv p a ) ( gcd p a ( isaprimetoneq0 is ) ~> 1 ). Proof. intros. intros P i. apply ( pr2 is ( gcd p a ( isaprimetoneq0 is ) ) ( pr1 ( gcdiscommdiv p a ( isaprimetoneq0 is ) ) ) ). intro t. apply i. destruct t as [ t0 | t1 ]. apply ii2. assumption. apply iii. rewrite <- ti. exact ( pr2 ( gcdiscommdiv p a ( isaprimetoneq0 is ) ) ). Defined.
```

```
Lemma primeandtimes ( p a b : hz ) ( is : isaprime p ) ( x : hzdiv p ( a * b ) ) : hdisj ( hzdiv p a ) ( hzdiv p b ). Proof. intros. apply ( primedivorcprime p a is ). intros j. intros P i. apply i. destruct j as [ jo | ji ]. apply iii. assumption. apply ii2. apply x. intro u. destruct u as [ ku ]. unfold hzdiv0 in u. set ( cd :=
```

```
bezoutstrong a p ( isaprimetoneq0 is ) ). destruct cd as [ cd f ]. destruct cd as [ c d ]. rewrite j1 in f. simpl in f. assert ( b ~> ( b * c + d * k ) * p ) as g. assert ( b ~> b * 1 ) as g0. rewrite hzmultri. apply idpath. rewrite g0. rewrite ( rngdistr hz ( b * 1 * c ) ( d * k ) p ). assert ( b * ( c * p + d * a ) ~> ( b * 1 * c * p + d * k * p ) ) as h. rewrite ( rngldistr hz ( c * p ) ( d * a ) b ). rewrite hzmultri. rewrite ! ( @rngassoc2 hz ). rewrite ( @rngcomm2 hz k p ). change ( p * k )%hz with ( p * k )%rng in u. rewrite u. rewrite ( @rngcomm2 hz b ( d * a ) ). rewrite ( @rngassoc2 hz ). apply idpath. rewrite <- h. rewrite f. apply idpath. intros Q uu. apply uu. split with ( b * c + d * k ). rewrite ( @rngcomm2 hz _ p ) in g. unfold hzdiv0. apply pathsinv0. assumption. Defined.
```

```
Lemma hzremaindermodprimeandtimes ( p : hz ) ( is : isaprime p ) ( a b : hz ) ( x : hzremaindermod p ( isaprimetoneq0 is ) a * b ~> 0 ) : hzremaindermod p ( isaprimetoneq0 is ) b ~> 0. Proof. intros. assert ( hzdiv p ( a * b ) ) as i. intros P i'. apply i'. split with ( hzquotientmod p ( isaprimetoneq0 is ) ( a * b ) ). unfold hzdiv0. apply pathsinv0. rewrite <- ( hzplusr0 ( p * hzquotientmod p ( isaprimetoneq0 is ) ( a * b )%rng ) )%hz. change ( a * b ~> ( p * hzquotientmod p ( isaprimetoneq0 is ) ( a * b )%rng + 0 )%rng ). rewrite <- x. change ( p * hzquotientmod p ( isaprimetoneq0 is ) ( a * b ) + hzremaindermod p ( isaprimetoneq0 is ) a * b ) with ( p * hzquotientmod p ( isaprimetoneq0 is ) ( a * b )%rng + ( hzremaindermod p ( isaprimetoneq0 is ) ( a * b )%rng ) )%hz. apply ( hzdivequationmod p ( isaprimetoneq0 is ) ( a * b ) ). apply ( primeandtimes p a b is i ). intro t. destruct t as [ t0 | t1 ]. apply t0. intros k. destruct k as [ k k' ]. intros Q j. apply j. apply iii. apply pathsinv0. apply ( hzqrtestr p ( isaprimetoneq0 is ) a k ). split. rewrite hzplusr0. unfold hzdiv0 in k'. rewrite k'. apply idpath. split. apply isreflhzleh. rewrite hzabsvalgh0. apply ( istranshzh0 _ 1 _ ). apply hzlhnsn. apply is. apply ( istranshzh0 _ 1 _ ). apply hzlhnsn. apply is. apply t1. intros k. destruct k as [ k k' ]. intros Q j. apply j. apply ii2. apply pathsinv0. apply ( hzqrtestr p ( isaprimetoneq0 is ) b k ). split. rewrite hzplusr0. unfold hzdiv0 in k'. rewrite k'. apply idpath. split. apply isreflhzleh. rewrite hzabsvalgh0. apply ( istranshzh0 _ 1 _ ). apply hzlhnsn. apply is. apply ( istranshzh0 _ 1 _ ). apply hzlhnsn. apply is. Defined.
```

```
Definition padiczero ( p : hz ) ( is : isaprime p ) := @rngunel1 ( commrnrngofpadicints p is ).
```

```
Definition padicone ( p : hz ) ( is : isaprime p ) := @rngunel2 ( commrnrngofpadicints p is ).
```

```
Lemma padiczero computation ( p : hz ) ( is : isaprime p ) : padiczero p is ~> setquotpr ( carryequiv p ( isaprimetoneq0 is ) ) ( @rngunel1 ( fpcommrnrng hz ) ). Proof. intros. apply idpath. Defined.
```

```
Lemma padicone computation ( p : hz ) ( is : isaprime p ) : padicone p is ~> setquotpr ( carryequiv p ( isaprimetoneq0 is ) ) ( @rngunel2 ( fpcommrnrng hz ) ). Proof. intros. apply idpath. Defined.
```

```
Lemma padicintssareintdom ( p : hz ) ( is : isaprime p ) ( a b : accmrnrngofpadicints p is ) : a # 0 -> b # 0 -> a * b # 0. Proof. intros p is. assert ( forall a b : commrnrngofpadicints p is, isprop ( ( pr1 ( padicpart p is ) ) a ( padiczero p is ) -> ( pr1 ( padicpart p is ) ) b ( padiczero p is ) -> ( pr1 ( padicpart p is ) ) ( padictimes p is a b ) ( padiczero p is ) ) ) as int. intros. apply impred. intros. apply impred. intros. apply ( pr1 ( padicpart p is ) ).
```

```
apply ( setquotuniv2prop _ ( fun x y => hProppair _ ( int x y ) ) ). intros a b. change ( pr1 ( padicpart p is ) ) ( setquotpr ( carryequiv p ( isaprimetoneq0 is ) ) a ) ( padiczero p is ) -> pr1 ( padicpart p is )
```

```

(setquotpr (carryequiv p (isaprimetoneq0 is)) b) (padiczero p is) ->
pri (padicpart p is) (padictimes p is (setquotpr (carryequiv p
(isaprimetoneq0 is)) a) (setquotpr (carryequiv p (isaprimetoneq0
is)) b)) (padiczero p is)). unfold padictimes. rewrite
padiczerocomputation. rewrite setquotprandpadictimes. rewrite 3!
padicpartcomputation. intros i j. apply i. introo i0. destruct i0
as [ i0 i1 ]. apply j. introo j0. destruct j0 as [ j0 j1 ]. rewrite
carryandzero in i1, j1. change (( @rngunel1 ( fpcommrnrng hz )) i0
) with 0%hz in i1. change (( @rngunel1 ( fpcommrnrng hz )) j0
) with 0%hz in j1. set ( P := fun x : nat => neq hz ( carry p (
isaprimetoneq0 is ) a x ) 0 ). set ( P' := fun x : nat => neq hz (
carry p ( isaprimetoneq0 is ) b x ) 0 ). assert ( isdecnatprop P
) as isdec1. intros m. destruct ( isdeceqhz ( carry p (
isaprimetoneq0 is ) a m ) 0%hz ) as [ l | r ]. apply ii2. intro
v. apply v. assumption. apply iii. assumption. assert ( isdecnatprop
P' ) as isdec2. intros m. destruct ( isdeceqhz ( carry p (
isaprimetoneq0 is ) b m ) 0%hz ) as [ l | r ]. apply ii2. intro
v. apply v. assumption. apply iii. assumption. set ( le1 :=
leastelementprinciple i0 P isdec1 ii ). set ( le2 :=
leastelementprinciple j0 P' isdec2 j1 ). apply le1. intro
k. destruct k as [ k' ]. apply le2. intro o. destruct o as [ o' o
]. apply total2tohexists. split with ( k + o )%nat.

assert ( forall m : nat, natlth m k -> carry p ( isaprimetoneq0 is )
a m > 0%hz ) as one. intros m m0. destruct ( isdeceqhz ( carry p (
isaprimetoneq0 is ) a m ) 0%hz ) as [ left0 | right0 ]. assumption.
assert empty. apply ( pr2 k' ) m m0. assumption. contradiction.
assert ( forall m : nat, natlth m o -> carry p ( isaprimetoneq0 is )
b m > 0%hz ) as two. intros m m0. destruct ( isdeceqhz ( carry p (
isaprimetoneq0 is ) b m ) 0%hz ) as [ left0 | right0 ]. assumption.
assert empty. apply ( pr2 o' ) m m0. assumption. contradiction.
assert ( dirprod ( neq hz ( carry p ( isaprimetoneq0 is ) a k ) 0%hz
) ( forall m : nat, natlth m k -> ( carry p ( isaprimetoneq0 is ) a
m ) > 0%hz ) ) as three. split. apply k'. assumption. assert (
dirprod ( neq hz ( carry p ( isaprimetoneq0 is ) b o ) 0%hz ) ( forall m : nat, natlth m o -> ( carry p ( isaprimetoneq0 is ) b m
) > 0%hz ) ) as four. split. apply o'. assumption. set ( f :=
hzfpstimesnonzero ( carry p ( isaprimetoneq0 is ) a ) k three o (
carry p ( isaprimetoneq0 is ) b ) four ). rewrite
carryandzero. change (( @rngunel1 ( fpcommrnrng hz )) ( k + o )%nat
) with 0%hz. rewrite carryandtimes.

destruct k. destruct o. rewrite <- carryandtimes. intros v. change (
hzremaindermod p ( isaprimetoneq0 is ) ( a 0%nat * b 0%nat ) > 0%hz
) in v. assert hfalso. apply ( hzremaindermodprimeandtimes p is ( a
0%nat ) ( b 0%nat ) v ). intros t. destruct t as [ t0 | t1 ]. apply
(pr1 k'). apply t0. apply ( pr1 o' ). apply t1. assumption.

intros v. unfold carry at 1 in v. change ( 0 + S o )%nat with ( S o
) in v. change ( hzremaindermod p ( isaprimetoneq0 is ) ( ( carry p
(isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) b ) ( S o
) + hzquotientmod p ( isaprimetoneq0 is ) ( precarry p (
isaprimetoneq0 is ) ( carry p ( isaprimetoneq0 is ) a * carry p (
isaprimetoneq0 is ) b ) o ) ) > 0%hz ) in v. change ( 0 + S o
)%nat with ( S o ) in f. rewrite f in v. change ( carry p (
isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) b ) with (
fpstimes hz ( carry p ( isaprimetoneq0 is ) a ) ( carry p (
isaprimetoneq0 is ) b ) ) in v. rewrite ( precarryandzeromult p is
a b 0%nat ( S o ) ) in v. rewrite hzgrand0q in v. rewrite hzplusr0
in v. assert hfalso. apply ( hzremaindermodprimeandtimes p is ( carry p
( isaprimetoneq0 is ) a 0%nat ) ( carry p ( isaprimetoneq0
is ) b ( S o ) ) ). assumption. intros s. destruct s as [ l | r ]. apply
k'. rewrite hzrandcarryr. assumption. apply o'. rewrite
hzrandcarryr. assumption. assumption. apply one. apply two. apply
natlthnsn.

intros v. unfold carry at 1 in v. change ( hzremaindermod p (
isaprimetoneq0 is ) ( ( carry p ( isaprimetoneq0 is ) a * carry p (
isaprimetoneq0 is ) b ) ( S + o )%nat + hzquotientmod p (
isaprimetoneq0 is ) ( precarry p ( isaprimetoneq0 is ) ( carry p (
isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) b ) ( k + o
)%nat ) ) > 0%hz ) in v. rewrite f in v. change ( carry p (
isaprimetoneq0 is ) a * carry p ( isaprimetoneq0 is ) b ) with (
fpstimes hz ( carry p ( isaprimetoneq0 is ) a ) ( carry p (
isaprimetoneq0 is ) b ) ) in v. rewrite ( precarryandzeromult p is
a b ( S k ) o ) in v. rewrite hzrand0q in v. rewrite hzplusr0 in
v. assert hfalso. apply ( hzremaindermodprimeandtimes p is ( carry p
( isaprimetoneq0 is ) a ( S k ) ) ( carry p ( isaprimetoneq0 is ) b
( o ) ) ). assumption. intros s. destruct s as [ l | r ]. apply
k'. rewrite hzrandcarryr. assumption. apply o'. rewrite
hzrandcarryr. assumption. assumption. apply one. apply two. apply
natlthnsn. Defined.

Definition padicintegers ( p : hz ) ( is : isaprime p ) : aintdom.
Proof. intros. split with ( acommrngofpadicints p is ). split.
change ( ( pri ( padicpart p is ) ) ( padicone p is ) ( padiczero p
is ) ). rewrite ( padiczerocomputation p is ). rewrite (
padiconecomputation p is ). rewrite padicpartcomputation. apply
total2tohexists. split with 0%nat. unfold carry. unfold
precarry. rewrite hzgranddir. rewrite hzrand0r. apply isnonzeroronghz.
apply padicintsareintdom. Defined.

Definition padics ( p : hz ) ( is : isaprime p ) : afld := afldfrac (
padicintegers p is ).

Close Scope rng_scope.
(** END OF FILE**)

```

References

- [1] M. Atiyah: Convexity and commuting Hamiltonians. *Bull. London Math. Soc.* **14** (1982) 1-15.
- [2] S. Awodey: Type theory and homotopy. To appear, on the arXiv as [arXiv:math/1010.1810v1](https://arxiv.org/abs/math/1010.1810v1), 2010.
- [3] S. Awodey, Á. Pelayo, and M. A. Warren: Voevodsky's univalence axiom in homotopy type theory, in preparation for *Notices Amer. Math. Soc.*
- [4] Y. Bertot and P. Castéran: *Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions.* Texts Theoret. Comput. Sci. EATCS Ser. Springer-Verlag, Berlin, 2004. xxvi+469 pp.
- [5] L. Brekke and P. G.O. Freund: p -adic numbers in physics. *Phys. Rep.* **233** (1993), no. 1, 1-66.
- [6] D. Bridges and F. Richman: *Varieties of constructive mathematics.* London Math. Soc. Lecture Note Ser., Cambridge University Press, 1987.
- [7] T. Delzant: Hamiltoniens périodiques et image convexe de l'application moment. *Bull. Soc. Math. France* **116** (1988) 315-339.
- [8] F. Gouvêa: *p -adic Numbers. An Introduction.* Universitext. Springer-Verlag, Berlin, 1993. vi+282 pp.
- [9] V. Guillemin and S. Sternberg: Convexity properties of the moment mapping. *Invent. Math.* **67** (1982) 491-513.
- [10] K. Hensel: Über eine Theorie der algebraischen Functionen zweier Variablen, *Acta mathematica* **23** (1900), no. 1, 339-416.
- [11] N. Koblitz: *p -adic Numbers, p -adic Analysis, and Zeta-Functions.* Second Edition. Grad. Texts in Math., 58. Springer-Verlag, New York, 1984. xii+150 pp.
- [12] R. Mines, F. Richman and W. Ruitenburg: *A course in constructive algebra.* Springer-Verlag, 1988.
- [13] Á. Pelayo and S. Vũ Ngoc: Semitoric integrable systems on symplectic 4-manifolds, *Invent. Math.* **177** (2009), 571-597.
- [14] Á. Pelayo and S. Vũ Ngoc: Constructing integrable systems of semitoric type, *Acta Math.* **206** (2011), 93-125.
- [15] Á. Pelayo and M. A. Warren: Homotopy type theory and Voevodsky's Univalent Foundations, submitted, on the arXiv as [arXiv:math/1210.5658](https://arxiv.org/abs/math/1210.5658), 2012.

- [16] W. H. Schikhof: *Ultrametric Calculus. An Introduction to p -adic Analysis.* Cambridge Stud. Adv. Math., 4. Cambridge University Press, Cambridge, 1984. viii+306 pp.
- [17] J.P. Serre: Classification des variétés analytiques p -adiques compactes. *Topology* **3** 1965 409-412.
- [18] V. Voevodsky: Coq library at www.math.ias.edu/~vladimir, 2011.
- [19] V. Voevodsky: Extended version of NSF proposal at www.math.ias.edu/~vladimir, 2010.

Álvaro Pelayo
 School of Mathematics
 Institute for Advanced Study
 Einstein Drive, Princeton
 NJ 08540 USA.

and

Washington University
 Mathematics Department
 One Brookings Drive, Campus
 Box 1146
 St Louis
 MO 63130-4899, USA.
E-mail:
apelayo@math.wustl.edu

Vladimir Voevodsky
 School of Mathematics
 Institute for Advanced Study
 Einstein Drive, Princeton
 NJ 08540 USA.
E-mail:
vladimir@math.ias.edu

Michael A. Warren
 School of Mathematics
 Institute for Advanced Study
 Einstein Drive, Princeton
 NJ 08540 USA.
E-mail:
mwarren@math.ias.edu